

AD-A146 444

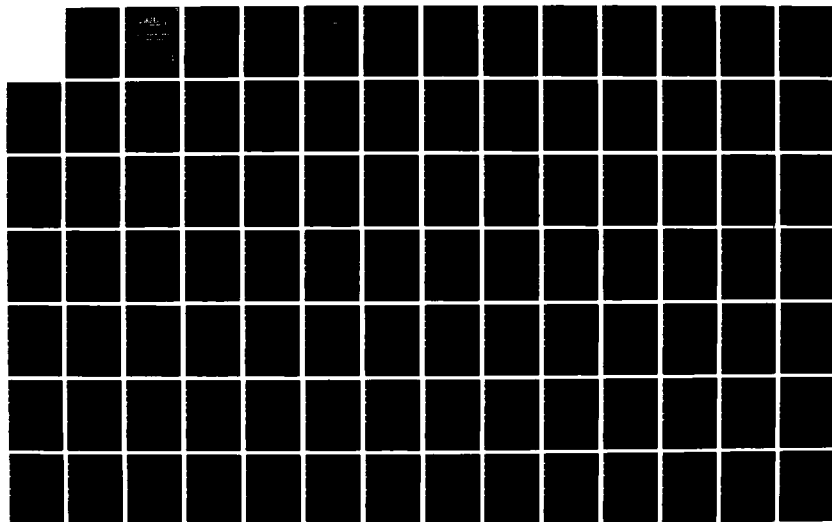
VLSI DESIGN TOOLS REFERENCE MANUAL RELEASE 20(U)  
WASHINGTON UNIV SEATTLE DEPT OF COMPUTER SCIENCE  
AUG 84 TR-84-88-07 MDA903-82-C-0424

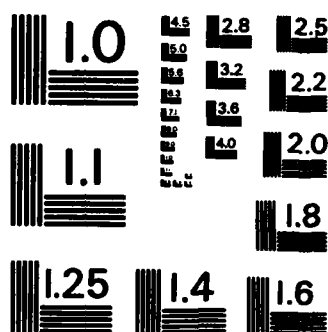
1/4

UNCLASSIFIED

F/G 9/5

NL





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A

12

AD-A146 444



# VLSI DESIGN TOOLS

## REFERENCE MANUAL

DTIC FILE COPY

Release 2.0  
AUG. 1, 1984

DTIC  
ELECTE  
OCT 04 1984  
S  
A  
E  
D

UW/NW VLSI Consortium  
University of Washington, Seattle, Washington 98195

This document has been approved  
for public release and sale; its  
distribution is unlimited.

84 09 27 018

unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR-84-08-07 ✓	2. GOVT ACCESSION NO. AD-A146444	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle)  VLSI DESIGN TOOLS REFERENCE MANUAL - RELEASE 2.0		5. TYPE OF REPORT & PERIOD COVERED Technical, interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s)  UW/NW VLSI Consortium		8. CONTRACT OR GRANT NUMBER(s)  MDA 903-82-C-0424
9. PERFORMING ORGANIZATION NAME AND ADDRESS UW/NW VLSI Consortium Department of Computer Science, FR-35 University of Washington Seattle, Washington 98195		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS  N/A
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA - IPTO 1400 Wilson Boulevard Arlington, Virginia 22209		12. REPORT DATE August 1984
		13. NUMBER OF PAGES 295
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) ONR University of Washington 315 University District Building 1107 N.E. 45th St., JD-16 Seattle, WA 98195		15. SECURITY CLASS. (of this report)  unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report)  Distribution of this report is unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) very large scale integration, VLSI design tools, CAD tools, CMOS, nMOS, VLSI layout, VLSI circuit simulation, design rule checking		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This report describes the use of the University of Washington/Northwest VLSI Consortium's package of VLSI design tools. The tools described are:  (see reverse)		



## APPENDIX A. TOOL DESCRIPTIONS

### 1. Functional Design Tools

- 1.1 "PEG": Translates a finite state machine description into logic equations.
- 1.2 "EQNTOTT": Converts logic equations into a truth table format.
- 1.3 "PRESTO": Minimizes truth table attributes.

### 2. Layout Tools

- 2.1 "FLAP": Pascal-based layout entry program.
- 2.2 "CAESAR": Graphical layout editor.
- 2.3 "TPACK": Library of C routines for constructing a layout by tiling.
- 2.4 "QUILT": Constructs a rectangular array of artwork tiles.
- 2.5 "TFLA": Technology independent pla generator.
- 2.6 "MKPLA": nMOS pla generator.

### 3. Display Tools

- 3.1 "CIFPLOT": Plots CIF designs using stipple patterns.
- 3.2 "PENPLOT": Makes penplots from CIF designs.
- 3.3 "VIC": Displays designs on TEK 4010 compatible devices and drives penplotters.

### 4. Rule Checkers

- 4.1 "DRC": Checks CIF designs against nmos (buried contact) rules.
- 4.2 "LYRA": Performs hierarchical design rule check.
- 4.3 "ERC": Checks nMOS design for consistent electrical properties.

### 5. Circuit Extractor

- 5.1 "MEXTRA": Extracts circuit description from CIF design.

### 6. Simulation Tools

- 6.1 "SPICE2G6": Device level circuit simulator.
- 6.2 "RNL": Event driven timing simulator.
- 6.3 "ESIM": Switch level simulator.
- 6.4 "CRYSTAL": Static timing verifier.
- 6.5 "POWEST": Estimates power consumption of an nMOS circuit.
- 6.6 "MTP": Displays RNL and SPICE output on a line printer.

### 7. Utilities/Miscellaneous

- 7.1 "NETLIST": Generates circuit description procedurally.
- 7.2 "PRESIM": Converts a circuit description to RNL input.
- 7.3 "PSPICE": Creates a complete spice input deck from a circuit description and user input.
- 7.4 "CELLIP": Layouts of a set of standard cells and pads in nMOS and CMOS.

# VLSI Design Tools

## Reference Manual

Release 2.0

8/1/84

TR-84-08-07



Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

**UW/NW VLSI Consortium**

Department of Computer Science

Room 315 Sieg Hall

University of Washington FR-35

Seattle, Washington 98195

## **TABLE OF CONTENTS**

- 1. Introduction**
  - 1.1 Who We Are : The UW/NW VLSI Consortium**
  - 1.2 Installation Instructions**
- 2. Overview of VLSI Design Tools**
  - 2.1 Enhancements since Release No. 1**
  - 2.2 Layout Tools : Functional Chart**
  - 2.3 Simulation Tools : Functional Chart**
  - 2.4 Tool Descriptions**
- 3. Manual Pages**
  - 3.1 UW VLSI Tools (including CMU, MIT, Boeing tools)**
  - 3.2 Berkeley tools modified at UW**
- 4. Procedural Layout Users Guides**
  - 4.1 PLAP User's Guide**
  - 4.2 Placement and Routing Procedures**
- 5. Standard Cell Library Guide**
- 6. RNL Users Guides and Tutorials**
  - 6.1 NETLIST User's Guide**
  - 6.2 PRESIM User's Guide**
  - 6.3 RNL User's Guide**
  - 6.4 NETLIST and RNL Tutorial for Beginners**
  - 6.5 NETLIST/PRESIM/RNL - A Tutorial**
- 7. SPICE User's Guide**

## **INTRODUCTION**

### **Who We Are : The UW/NW VLSI Consortium**

UW/NW VLSI Consortium members include the University of Washington, represented by the Computer Science Department, and five Pacific Northwest firms: Boeing Aerospace, John Fluke Manufacturing, Tektronix, Honeywell Marine Systems, and Microtel Pacific Research of Canada. The purpose of the Consortium is to advance the state of the art in VLSI technology and to transfer this technology to American industry.

Each corporate member of the Consortium has a full-time liaison on campus who cooperates with faculty members on circuit design as well as filling the role of visiting faculty, working with graduate students and contributing valuable "real world" experience to the Department of Computer Science. This program is serving as a demonstration of cooperative research and technology exchange among universities and industry.

The Consortium maintains a VLSI design system and plans to evolve its capability over time, as well as to provide training in its use. The Consortium validates the system by exercising it with industry design problems. Missing pieces will be identified for guiding future research and development. The resulting system is made available to other universities and industry.

Students from industry and universities are being trained in the VLSI design methodology through a series of regularly scheduled intensive courses. Educational materials are being developed to allow similar training capability to be distributed to other locations throughout the country. Refresher courses, symposia and reports are being scheduled as new capabilities are added.

### Installation Procedure

The distribution tape contains VLSI design tools that run on a VAX with Berkeley 4.1 UNIX. Many of them will not run on other machines or other versions of UNIX. In addition, VLSI display tools require plotting or graphics devices:

Pen plotters (HP7221, HP7580)

Dot matrix printers (Versatec, Varian, Printronix)

Interactive graphics devices with bitpads (AED512, Metheus Omega 440)

The tape contains two 1600-bpi tar format files:

- 1) UW/NW VLSI Design Tools, Release 2.0.
- 2) 1983 VLSI Tools as distributed by Berkeley.

The first file includes complete, self-contained tools (e.g. PLAP) as well as modifications to several Berkeley tools contained in the second file. Organization of the first file is as follows:

bin	executables and shell scripts
doc	user manuals
include	source header files for some tools
lib	archived libraries and other stuff
man	UNIX programmer's manual entries for each tool
src	source code, make files, and installation scripts

To install the tools:

1. Copy the first file on the tape to a suitable directory (on our system it's /src/vlsi/vlsi-tools) using the `tar` command. If you do not already have the 1983 Berkeley Tools you will need to copy the second file of the tape into some other directory.
2. Set environment variable `UW_VLSI_TOOLS` to the full path name of this directory (e.g. `setenv UW_VLSI_TOOLS /src/vlsi/vlsi-tools`). (Put this in your login file.)
3. Set (login file again) the following environment variables:

`PATH` to some path that includes `$UW_VLSI_TOOLS/bin`  
`PLAPPATH` to some path that includes `$UW_VLSI_TOOLS/lib/technology`  
`RNLPATH` to some path that includes `$UW_VLSI_TOOLS/lib/rnl`

Directory `$UW_VLSI_TOOLS/bin` should precede `/usr/ucb` and `/usr/bin` if you want to take advantage of the new `man` command that accesses `$UW_VLSI_TOOLS/man`, `~cad/man` (Berkeley VLSI tools), and `/usr/man`.

4. Cd to `$UW_VLSI_TOOLS/src` and run 'MAKE man' to initialize the manual pages.
5. If you are running Berkeley 4.2 you will have to run `MAKEALL` (in `$UW_VLSI_TOOLS/src`) to recompile everything. Most of the tools have been run successfully on 4.2, but you should eliminate the `-Dbad41` flag in `src/lib/libutil/makefile` when compiling on 4.2.

There are several improvements (including man pages) and bug fixes for the Winter 83 Berkeley tools available in `$UW_VLSI_TOOLS/src/ucb-cad`. These may be installed using the `INSTALL-UCB` script in `$UW_VLSI_TOOLS/src`. You may wish to review the `README` file in `$UW_VLSI_TOOLS/src/ucb-cad` and edit `INSTALL-UCB` appropriately if you want only some of the changes. Since only the changed source files are included in `$UW_VLSI_TOOLS/src/ucb-cad`, you must follow the procedures documented with the Berkeley VLSI tools to complete the installation after using `INSTALL-UCB`.

The following miscellaneous sources are included:

Sources for `lpr` that complement the changes to `cifplot` and allow `cifplot` to be used with a Printronix.

## Introduction

UW/NW VLSI Consortium

Sources for a parallel interface device driver for a Methuen omega 440 color graphics display.  
A DEC DR-11W is needed in addition to the Methuen.

## 2.0 OVERVIEW OF VLSI DESIGN TOOLS

### ENHANCEMENTS SINCE RELEASE NO. 1

The first release of the UW/NW VLSI Design System fully supported only nmos processes. This design system release supports both the GTE 5 micron isoplanar CMOS process and the MOSIS 3 micron bulk CMOS process as well as MOSIS nmos processes. Technology files are employed wherever possible to isolate technology dependence (see the man page titled "technology").

We have made a number of enhancements to the tools since the first release. The pascal layout artwork program PLAP can now write Manhattan designs in the cecar database format, so that low level cell design can be done with cecar and combined with the procedural approach of PLAP. A library of PLAP routines for placement and interconnect of cells as well as larger modules is included (See the document "Placement and Routing Procedures").

Layouts for a set of standard cells implementing simple logic functions are available in both GTE isoplanar CMOS and MOSIS bulk CMOS. These cells, as well as a set of I/O pads and padframes, are described in the Standard Cell Library Guide. PLA templates in both CMOS technologies may be used in conjunction with the Berkeley program TPLA to generate PLA's.

A new display program VIC (View Integrated Circuits) will display layouts on terminals supporting Tektronix 4010 vector graphics protocol. This program provides a convenient interface for obtaining a hardcopy through penplotters. It supports the hierarchy of the design and has a windowing capability to improve resolution.

We have modified the programs that use the ".sim" file circuit description, eliminating the inconvenience of having to convert between the MIT and Berkeley formats. Programs that read the ".sim" file, such as crystal, sim2spice and presim now read either format (see the functional chart of the simulation tools and the man page titled "simfile").

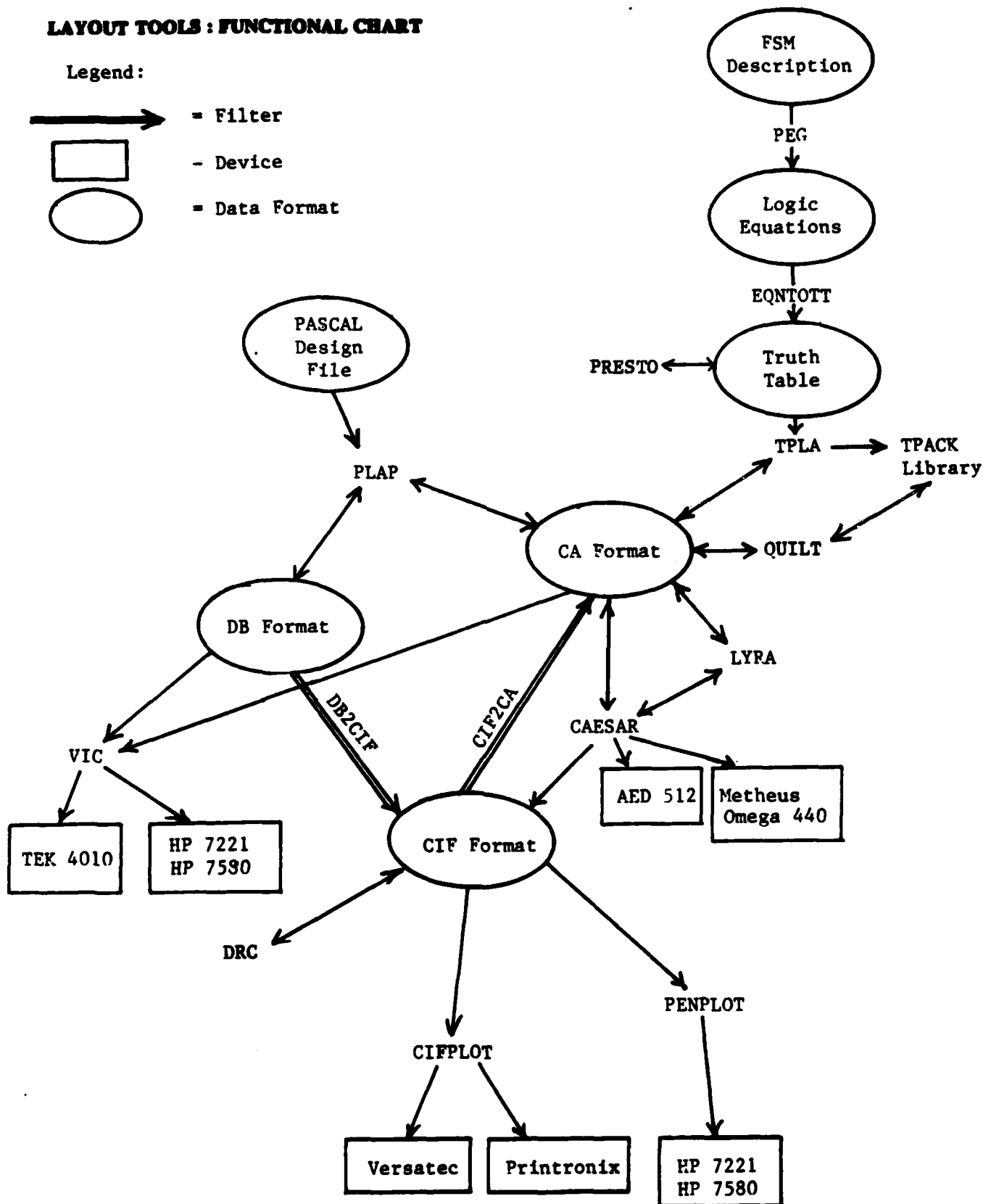
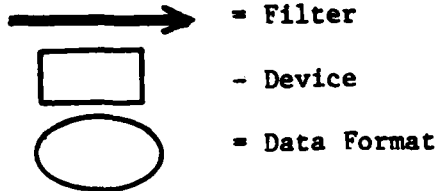
The event-driven simulator RNL (written by Chris Terman of MIT) has been considerably enhanced (See the RNL Users Guides). Additional primitives have been added so that the RNL interface would be more consistent with most versions of Lisp. A recoding of the Lisp interpreter allowed node status to be changed far more efficiently, resulting in a 10-fold decrease in simulation executions for large designs. Finally, a "patterns package" was added so that complex and repetitive digital waveforms could be generated far more easily. The recently written program MTP (Multiple Time-series Program) displays RNL and SPICE waveforms on a Printronix printer.

Because of the complexity of the command syntax of RNL, we have developed several tutorial aids. The "NETLIST and RNL Tutorial for Beginners" takes a 10 bit shift register from inception in NETLIST, to simulation with RNL, and finally to waveform plotting with MTP. The "NETLIST/PRESIM/RNL - A Tutorial" is intended to illustrate the flexibility of the more advanced features of RNL.

## 2.0 OVERVIEW OF VLSI DESIGN TOOLS

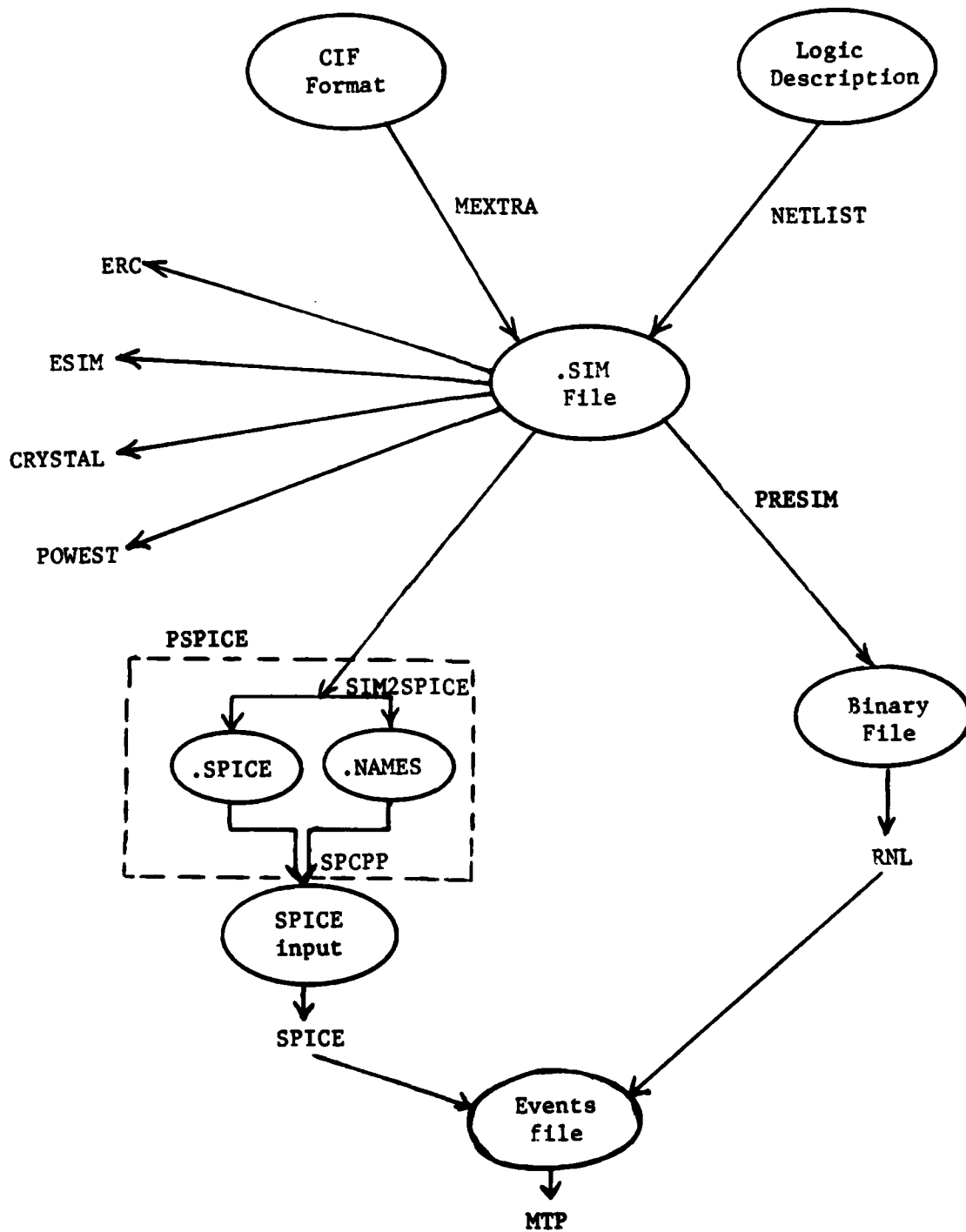
## LAYOUT TOOLS : FUNCTIONAL CHART

Legend:





## SIMULATION TOOLS : FUNCTIONAL CHART



**TOOL DESCRIPTIONS**

The following is a brief overview of the vlsi-tools we are distributing. An asterisk (\*) appears after the name of tools that are part of the Berkeley Distribution.

The functional design tools translate high level design descriptions into layout tool input. Layout tools are used to design the actual artwork for the circuit; display tools are used to display circuit designs. Electrical and design rule checkers are included as well as a variety of timing and logic simulation tools.

**Functional Design Tools**

Translate high level design descriptions into layout tool input.

- peg* \* Translates a language description of a finite state machine into logic equations compatible with *eqntott*. ( Gordon Hamachi, UCB )
- eqntott* \* Converts logic equations into a truth table format to be used as input to *mkpla* or *tpla*. ( Bob Cmelik, UCB )
- presto* Tries to minimize the number of product terms in a truth table ( UCB ).

**Layout Tools**

Design artwork for circuit.

- plap* Compiles a pascal design file created by the user, links it to a package of artwork generation routines, and executes it to produce artwork data files in either db or Caesar formats. Handles general nonmanhattan designs. ( Bruce Yanagida, Boeing )
- caesar* \* A powerful display editor for manhattan designs written at Berkeley. Runs on AED512 or Metheus Omega 440 color displays. Requires a design to be inputted in *caesar* format but will optionally output in CIF. ( John Ousterhout, UCB )
- tpack* \* A collection of C routines that assembles Caesar-generated artwork tiles into semi-regular design modules. This is not an executable program but a library of routines. Employed by *quilt* and *tpla*. ( Robert Mayo and John Ousterhout, UCB )
- quilt* \* Assembles Caesar-generated artwork tiles into a rectangular array. ( Robert Mayo, UCB )
- tpla* \* Technology independent artwork generator. Employs Caesar-generated artwork tiles and a truth table. ( Robert Mayo, UCB )
- mkpla* \* Generates NMOS PLA artwork from a truth table. This program has been largely superseded by *tpla*. ( Howard A. Landman, UCB )

**Display Tools**

Display circuit designs.

- cifplot* \* Berkeley program that plots a CIF design in stipple patterns on Versatec or Printronix dot-matrix printers. ( Dan Fitzpatrick, UCB )
- penplot* Penplotting programs for HP 4- and 8-pen plotters. Compatible with either db or CIF formatted designs. ( Boeing )
- vic* Display program for a Tektronix 4010 compatible device with penplot options for HP 4- and 8-pen plotters. Takes db or Caesar files. ( Bruce Yanagida, Boeing; Larry McMurchie, UW/NW VLSI Consortium ).

**Rule Checkers**

Geometric and electrical rule checking.

- drc** Design rule checker from Carnegie-Mellon. Checks MOSIS NMOS (buried contact) rules on Manhattan CIF designs. ( Dorothea Haken, CMU )
- drcscript** Merges the design rule violation files created by *drc* with the CIF design file for display purposes. ( Dorothea Haken, CMU )
- lyra \*** Performs hierarchical design rule check on a Caesar-formatted design using a corner based algorithm. NMOS, CMOS and user-defined rulesets are available. ( Michael Arnold, UCB )
- erc** Checks circuit description of an nmos design for consistent electrical properties. ( Boeing )

**Circuit Extractors**

Extract simulation database from layout database.

- mextra \*** Extracts a .sim file from a CIF input file. *Mextra* provides input files for *erc*, *esim*, *crystal*, *spice* and *rn1*. ( Dan Fitzpatrick, UCB )

**Simulation Tools**

Logic and timing simulation.

- spice2g6** The well known device level circuit simulator with minor mods to output an additional file that allows multiple time series plots to be made. ( Lawrence Nagel, UCB )
- rn1** An event driven "timing" simulator. It uses logic levels and a simplified circuit model to estimate timing delays through digital circuits. It also has a mode that allows it to be used as a switch (gate) level simulator. ( Chris Terman, MIT )
- esim \*** An NMOS gate level simulator. It uses logic levels and models transistors as perfect switches. ( Chris Terman, MIT )
- crystal \*** A static timing verifier. It uses a simplified circuit model to estimate the worst case delay through a circuit. ( John Ousterhout, UCB )
- powest \*** Reads an nmos circuit description in the format required by *esim* and writes estimates of its DC power requirements.
- mup** Displays, a multiple time series plots on a Printronix printer, the simulation output from either *spice* or *rn1*. (William Beckett, UW/NW VLSI Consortium).

**Filters and Utilities**

Filters to convert from one database to another and useful utilities.

- db2cif** Converts from db format (produced by a PLAP program) to CIF format. ( Boeing )
- cif2ca \*** Converts from CIF format to Caesar format. ( Peter Kessler, UCB )
- ca2db** Converts from Caesar format to db format. ( Bruce Yanagida, Boeing )
- netlist** Generates circuit descriptions (net lists). The output is a .sim file. ( Chris Terman, MIT ).
- presim** Converts a .sim file into the binary format required by *rn1*. In the process *presim* simplifies the circuit by identifying gates in the circuit. ( Chris Terman, MIT )
- pspice** Runs *sim2spice* and *spcpp*. In addition to running those programs it concatenates various files so as to create a complete Spice input deck. This reduces considerably the effort of simulating variations on a particular circuit. ( Rob Fowler, UW/NW VLSI Consortium )

- sim2spice* \* Reads a *.sim* file containing a description of a circuit and writes a *.names* file and a *.spice* file. The former contains a translation file from node names in the *.sim* file to the Spice node numbers. The *.spice* file contains a description of the devices in the circuit in a form acceptable to Spice. ( Dan Fitzpatrick, UCB )
- spcpp* Facilitates the writing of Spice input by allowing the user to refer circuit nodes using mnemonic node labels rather than Spice node numbers. ( Rob Fowler, UW/NW VLSI Consortium )
- spice* A *csh* script for running *spice2g6*. ( Lawrence Nagel, UCB )

**NAME**

**ca2db** - convert a Caesar cell to equivalent db (PLAP) files

**SYNOPSIS**

**ca2db** [*options*] *symbolname*

**DESCRIPTION**

*Ca2db* converts any Caesar-created cell (*symbolname.ca*), together with all its children, to the db data base format (*symbolname.sym*, *symbolname.att*) used by PLAP programs. (Note that a PLAP program is able to read .ca files, and hence the utility of *ca2db* is to some degree obviated.)

To facilitate the interconnection of a Caesar-created cell by a PLAP program, Caesar labels are also translated into attributes in the *symbolname.att* file. Each Caesar label produces two attributes:

*labelname\_x* The *x* location of *labelname*

*labelname\_y* The *y* location of *labelname*

Therefore, if the ports of a Caesar cell are labeled, it is trivial to use the PLAP *fa* (fetch attribute) command to get the port locations.

The *options* are as follows:

**-f** *directory*

*Directory* contains the .ca files. Default is the current directory.

**-t** *directory*

Db files are written into *directory*. Default is the current directory.

**-e**

include *lyra* design rule error files in the translation. This option allows the user to view design rule errors produced by *lyra* even if *caesar* is not available.

**DIAGNOSTICS**

If an error occurs, a message is written to standard error and the program exits with a non-zero status. *Ca2db* assumes the Caesar data base is intact, so don't mess with those files!

**FILES**

*symbolname.ca*  
*symbolname.ly*  
*symbolname.att*  
*symbolname.sym*

**SEE ALSO**

*caesar*(cad1)  
*plap*(1.vlsi), *penplot*(1.vlsi)

*PLAP User's Guide*, VLSI Design Tools Reference Manual, UW/NW VLSI Consortium, University of Washington.

*Editing VLSI Circuits with Caesar*, John Ousterhout, University of California, Berkeley.

**AUTHOR**

Bruce A. Yanagida

**NAME**

**db2cif** - convert a db (PLAP) cell into an equivalent CIF file

**SYNOPSIS**

**db2cif** [ *options* ]

**DESCRIPTION**

*Db2cif* creates a complete CIF file for a specified symbol in the db database. Upon invocation, *db2cif* will prompt for the db database input file (extensions are not necessary and the tilde character is allowed). The response should be the pathname of the *symbolname* for which a CIF file is desired. For example, if the db library, *libraryname* is in the current directory, then the response should be *libraryname/symbolname*.

input file ? *libraryname/symbolname*

The output file, *symbolname.cif*, is created in the current working directory.

If a symbol is out of date (it contains other symbols that have been created more recently) an error message is produced and, after the entire database has been checked, *db2cif* aborts without creating a CIF file (see -i option below).

The options are:

- i Specifies that a CIF file is to be created in spite of any out-of-date symbol errors. Error messages will not be suppressed and a comment to the effect that out-of-date symbols were encountered will be included in the CIF output.
- i n Selects the centimicrons to lambda ratio to be n. Default is 200.

**FILES**

*libraryname/symbolname.att*  
*libraryname/symbolname.sym*  
*symbolname.cif*

**SEE ALSO**

*plap(1.vlsi)*  
*cifplot(1.vlsi)*, *mextra(1.vlsi)*, *alldrc(1.vlsi)*, *fastdrc(1.vlsi)*

**AUTHOR**

Bruce A. Yanagida (Boeing Aerospace)

**BUGS**

*Db2cif* will not work if executed within the *libraryname* directory. *Libraryname* is required in the input file pathname and therefore must *not* be the current directory.

**NAME**

**drc, drcscript** - NMOS VLSI layout design rule checker

**SYNOPSIS**

**drc** [-kn] *basename.cif* [*lambda*]

**drcscript** *basename.cif*

**DESCRIPTION**

**Drc** analyzes an NMOS CIF file for geometric rule violations using MOSIS (buried contact) rules. It is limited to rectilinear, orthogonal geometry. Wires are taken apart into rectangles, and round flashes are approximated by squares. Polygons and non-manhattan rectangles are simply ignored.

The options are as follows:

**-k** Keep around all intermediate files.

**-n** Keep around files of unfiltered error messages.

**lambda** Specify new lambda to be *lambda* microns. Default is 2.0 microns.

For large files, **drc** should be run in batch mode (an 8000 transistor chip takes 2.4 11/780 cpu hours).

When **drc** finds violations, it creates CIF files of rectangles marking the geometric edges involved. These markers are placed on the glass layer. Separate files are created for each class of error, named *err.errortype.basename*.

To abort **drc**, hit the **BREAK** key and wait while it outputs some error messages until it eventually quits.

**Drcscript** will merge **drc** output files, labels indicating error type, and the original CIF file into a single file, *drcbasename.cif*. If this file is processed by *cif2ca* the results may be viewed with *caesar*. Errors show up as orange boxes in the glass layer. Each pair of boxes involved in an error will have an associated *errortype* label which will be located at the midpoint between the centers of the two boxes.

NMOS rules checked by **drc**:

Errortype	Rule	Lambda
dS	diffusion spacing	3.0
iOg	implant-gate overlap	1.5
iSg	implant-gate spacing	2.0
pS	poly spacing	2.0
pOg	poly-gate overlap	2.0
pSd	poly-diff spacing	1.0
cS	cut-cut spacing	2.0
dcSg	diff-cut to gate	2.0
mW	metal width	3.0
iNOg	implants with no gates	
XC	cuts with no D or P	
dW	diffusion width	2.0
ntdW	non-xtr diff width	2.0
iS	implant-implant	1.5
pW	poly width	2.0
gW	gate width	2.0
cW	cut min width	2.0
cL	cut max length	6.0
mOc	metal-cut overlap	1.0
dOc	diff-cut overlap	1.0

pOc	poly-cut overlap	1.0
aBP	buried-poly spacing	2.0
aBD	buried-diffusion spacing	2.0
oBD	buried-diffusion overlap	2.0
oBU	buried-contact surround	1.0
bW	buried-contact width	2.0

**SEE ALSO**

caesar(cad1), cif2ca(1.vlsi), penplot(1.vlsi)

*A Geometric Design Rule Checker*, Dorothea Haken, VLSI Document V053, Carnegie Mellon, 9 June 1980.

**FILES**

basename.cif  
err.errortype.basename  
drcbasename.cif  
errbasename.cif

**AUTHOR**

Dorothea Haken (CMU)

**BUGS**

The poly-overlap-gate check fails when the overlap is exactly zero.

Spacing checks do not consider mutual connectivity. Sometimes weird things will happen near butting contacts.

Cuts in diffusion or poly that do not have metal covering are not reported.

Diagonal spacing checks do not consider the true diagonal distance.



**NAME**

**erc** - VLSI (nMOS) electrical rules checker

**SYNOPSIS**

```
erc [-abedeghlmrstvw] [-l float] [-n n] basename  
ercscript basename.cif
```

**DESCRIPTION**

*Erc* reads a switch level circuit description of an nMOS chip produced by *mextra* and a designer-supplied list of inputs and outputs, and produces a listing of possible electrical rule violations, a cross-reference of nodes and transistors, and CIF code to display the locations of the electrical rules violations with *cifplot*, *penplot* or *caesar*. (Before *caesar* is invoked the filter program *cif2ca* must be run on the *erc* cif output file.)

*Erc* will read the files *basename.slm*, *basename.lnb*, *basename.nodes*, *basename.ai* and create up to five new files: *basename.erc*, *basename.crf*, *basename.err*, *ercbasename.cif* and *ercbasename.err*. If *basename* is greater than seven characters in length only the first seven characters will be used for *basename* in the files *ercbasename.err* and *ercbasename.cif*.

*Erc* recognizes many command line options. These can be used to enable or disable certain tests and to specify whether a cross-reference will be generated. Several options may be selected. A summary of the options in effect is placed at the beginning of the file *basename.erc*. A dash (-) may precede each option specifier or options may be concatenated together with a single leading dash. The following is a list of options that may be included on the command line:

- a Disables checking for transistors with the same source and drain node.
- b Disables the "pass transistor gating pass transistor" check.
- c This option causes the file *basename.crf* to be generated. This file contains a cross-reference of all transistors and nodes. The format of this file is discussed later.
- d Causes all depletion mode transistors that are not attached to *Vdd* to be listed.
- e This option causes all enhancement mode transistors attached to *Vdd* to be listed.
- g The "transistor reachable from ground" test is disabled.
- h Disables listing of badly formed transistors.
- i Disables listing of transistors gated by *GND*.
- j Disables listing of transistors gated by *Vdd*.
- l float A floating point value is given. This becomes the lower limit for the k-ratio test. Default is 4.000.
- m Disables "multiple pullup transistors on the same node" test.
- n Causes explicit listing of the locations of all unconnected nodes instead of giving the count only.
- r Disables ratio checking.
- s Disables listing of *Vdd-GND* shorts across transistors.
- t Causes explicit listing of the locations of all unconnected transistors instead of giving the count only.
- n n The integer *n* specifies cif units per lambda ( 250 is the default ).
- v This option disables the "reachable from *Vdd*" test.
- w Disables the "pullups separated by only 1 pass transistor" test.

**INPUT FILES***basename.stm**basename.ai**basename.nodes*

These files are obtained by running *mextra*. No modifications to these files are required for *erc*. It is important to note that *erc* understands only *Vdd* for power and *GND* for ground.

*basename.lab*

There is one line in *basename.lab* for each input and output of the circuit being checked. The format for each line is as follows:

[non-restored|restored] input|output|bidirectional [pad] nodename

The arguments enclosed by [] are optional. Alternatives are separated by '|'. *Nodename* is the name of the input or output.

**OUTPUT FILES***basename.erc*

This file contains a summary of the tests *erc* applied to the circuit, a breakdown of the number of transistor types, and warnings of any potential errors *erc* found.

The first page summarizes the tests applied to the circuit and has a count of transistors acting as pullups, pulldowns, precharges, lightning arrestors, internal pulldown and pass transistors, and transistors that are errors by themselves. After this first page any errors that *erc* detected are listed. Each different type of error starts a new page of the output.

*basename.crf*

*Baseline.crf* contains a cross-reference for all transistors and nodes. *Baseline.crf* is generated with the *-c* option.

The first section has a numbered entry for each transistor in *basename.stm*. The entry consists of type (enhancement or depletion), gate, source, and drain nodes, length and width of the channel in lambda, and the x and y coordinates in lambda.

Example of a transistor entry:

[1] e latch 67 Vdd 2.0 2.0 9.0 17.5

This is transistor #1. It is enhancement mode. Its gate, source and drain nodes are *latch*, 67 and *Vdd* respectively. This transistor is 2 lambda by 2 lambda and is located at (9.0,17.5) relative to the origin.

The second section of *basename.crf* has an entry for each node in the circuit. An entry consists of the node's name, its x and y coordinate, input or output type, a list of transistor gates the node is attached to, and a list of transistor sources and drains that this node is connected to.

Example of a node entry:

[SR.in 14.5 4.0] G: 6 S&D: 17 12

This node's name is *SR.in*. (14.5,4.0) is a point located on this node. The *i* after the x and y coordinates means this is an input. When the characters *b*, *i*, *n*, and *o* are in this field they mean *bidirectional*, *input*, *none* (not an *input* or *output*), and *output* respectively. After *G*: there is a list of transistor numbers that are gated by this node (there may be none). After *S&D*: there is a list of transistors that have their source or drain attached to this node. The transistor numbers in the node entries correspond to the transistor numbers in the first section of this file.

**ercbasename.cif**

This file should be used with *penplot* or *ceasar* as an aid in debugging the circuit. It contains the original CIF file together with a symbol that has glass layer boxes around possible errors in the circuit. The symbol that contains the original CIF file and the error layer is named *ercmerge* and the symbol containing the errors only is called *ercerrors*. All other symbol names remain unchanged. At the center of each error box there is a label that denotes the type of error detected. A list of these labels and the type of error they denote follows:

<b>E_PAD</b>	This pad is the source or drain of a pass transistor.
<b>E_RAT</b>	The <i>k</i> -ratio for this pullup is less than the allowed lower limit (default limit is 4.000).
<b>E_MPU</b>	This node has more than one pullup attached to it.
<b>E_PPU</b>	Two nodes with pullups are separated by one pass transistor.
<b>E_VDD</b>	This transistor is not reachable from <i>Vdd</i> .
<b>E_GND</b>	This transistor is not reachable from <i>GND</i> .
<b>E_PUD</b>	This pullup is not depletion mode.
<b>E_PDE</b>	This pulldown is not enhancement mode.
<b>E_ERR</b>	This transistor's nodes are nonsense.
<b>E_PGP</b>	This pass transistor is gated by another pass transistor.
<b>E_SME</b>	This transistor has the same source and drain nodes.
<b>E_AON</b>	This transistor is always on.
<b>E_AOF</b>	This transistor is always off.
<b>E_SRT</b>	There is a power to ground short from the source to the drain of this transistor.
<b>E_TNC</b>	This transistor is not connected to anything.
<b>E_NNC</b>	This node is not connected to anything.

**ercbasename.err**

This file contains a CIF symbol containing only the errors *erc* found. Use this file to look at only the error boxes.

**basename.err**

This file contains the CIF code that is destined to become the symbol *ercerrors* in the file *ercbasename.cif* and *ercbasename.err*. This file is only meant to be used if *ercbasename.cif* or *ercbasename.err* has been inadvertently removed. To recreate the merged representation of the CIF file type:

```
ercscript basename.cif
```

This will make two files, *ercbasename.cif* and *ercbasename.err*.

**FILES**

```
basename.sim
basename.al
basename.nodes
basename.lab
basename.erc
basename.cif
ercbasename.cif
ercbasename.err
basename.err
```

**SEE ALSO**

```
cifplot(1.vlsi), mextra(1.vlsi), drc(1.vlsi)
```

**AUTHOR**

Keith Pennick (Boeing Aerospace)

**BUGS**

Doesn't check k-ratios for pulldown networks over 1-level deep.

Multiple errors at the same node or transistor make the error messages displayed by the display programs hard to read.

CIF resolution for the error symbol *ercerrors* in *ercbasename.cif* is only 1 lambda. This sometimes results in a label not on any layer. The corresponding error message in *basename.arc* has the exact coordinates.

**NAME**

laytools - Introduction to VLSI layout tools

**DESCRIPTION**

This document is an introduction to the tools in the UW/NW-VLSI Consortium collection that are used in connection with the layout of VLSI circuits. These fall into four categories:

- |                                |   |
|--------------------------------|---|
| <i>functional design tools</i> | which translate high level design descriptions into layout tool input;  |
| <i>layout tools</i>            | which are used to design the actual artwork for the circuit;            |
| <i>display tools</i>           | which are used to display circuit designs;                              |
| <i>rule checkers</i>           | which are used to check geometric and electrical parameters of designs; |

The functional design tools are:

- |                |   |
|----------------|---|
| <i>peg</i>     | Translates a language description of a finite state machine into logic equations compatible with <i>eqn2otl</i> . |
| <i>eqn2otl</i> | Converts logic equations into a truth table format to be used as input to <i>mkpla</i> or <i>tpla</i> .           |
| <i>presto</i>  | Tries to minimize the number of product terms in the truth table.   |

The layout tools are:

- |               |   |
|---------------|---|
| <i>plap</i>   | Compiles a pascal design file created by the user, links it to a package of artwork generation routines, and executes it to produce artwork data files in either db or Caesar format. Handles general nonmanhattan designs. |
| <i>db2cif</i> | Converts from db format (produced by a PLAP program) to CIF format.   |
| <i>cif2ca</i> | Converts from the CIF format to Caesar format.  |
| <i>caesar</i> | The renowned graphical editor of manhattan designs written at Berkeley. Requires a design to be in Caesar format but will optionally output in CIF.   |
| <i>tpack</i>  | A collection of C routines that assembles Caesar-generated artwork tiles into semi-regular design modules. This is not an executable program but a library of routines. Employed by <i>quilt</i> and <i>tpla</i> .          |
| <i>quilt</i>  | Assembles Caesar-generated artwork tiles into a rectangular array.  |
| <i>mkpla</i>  | Generates NMOS PLA artwork from a truth table. This program has been largely superseded by <i>tpla</i> .  |
| <i>tpla</i>   | Technology-independent artwork generator. Employs Caesar-generated artwork tiles and a truth table.   |

The display tools are:

- |                |  |
|----------------|--|
| <i>cifplot</i> | Berkeley program that plots a CIF design in stipple patterns on Versatec or Printronix dot-matrix printers.  |
| <i>penplot</i> | Penplotting programs for HP 4- and 8-pen plotters. Compatible with either db or CIF formatted designs.   |
| <i>vic</i>     | Display program for a Tektronix 4010 compatible device with penplot options for HP 4- and 8-pen plotters. Compatible with either db or Caesar formats. |

The rule checkers are:

- drc* Design rule checker from Carnegie-Mellon. Checks MOSIS nmos (buried contact) rules on Manhattan CIF designs.
- drcscript* Merges the design rule violation files created by *drc* with the CIF design file for display purposes.
- lyra* Performs hierarchical design rule check on a Caesar-formatted design using a corner based algorithm. NMOS, CMOS and user-defined rulesets are available.

#### COMMENTS

There are two basic layout programs - *plap* and *caesar*. *Caesar* requires the availability of a graphics display such as an AED512 or Metheus Omega 440. It also restricts one to Manhattan designs. The PLAP language has no such restrictions. However, using PLAP to enter geometries of low level cells is somewhat tedious. Once a design is in either db or Caesar format, conversion to CIF is done through *db2cif* or by a Caesar command. With the design in CIF, a circuit description for simulation may be extracted using *maxtra* (see the "simtools" manual entry).

**NAME**

**mexnodes** - integrate intermediate nodes extracted by *mextra* with the original *caesar* design.

**SYNOPSIS**

**mexnodes** [*options*] *basename*

**DESCRIPTION**

*Mexnodes* is a shell script that uses *cif2ca* and *caesar* to generate a Caesar-format file. This file allows the user to view the intermediate nodes named by *mextra* on the original design. *Mexnodes* can be helpful when a simulation tool reports errors at a node not named by the user, as such errors are sometimes hard to locate. The output file created by *mexnodes* is named *mxbasename.ca*. This file can be then viewed using *caesar* in order to find a given node.

The *options* are as follows:

**-t technology**

*Technology* is one of *nmos*, *isocmos*, or *cmospw*. Default is *nmos*.

**-l lambda**

*Lambda* specifies the centimicrons to lambda correspondence of the design. Default is 200 centimicrons per lambda.

**FILES**

*basename.ca*  
*mxbasename.ca*  
*basename.nodes*  
*basename.cif*

**SEE ALSO**

*caesar*(1), *cif2ca*(CAD), *mextra*(CAD1)

**AUTHOR**

Terry J. Ligocki

**BUGS**

**NAME**

**mtp** - Multiple Time-series Plot for simulator output

**SYNOPSIS**

**mtp** *behavior-file directive-file plot-file*

**DESCRIPTION**

*Mtp* plots the output of *rnl* and *spice* simulations on the Printronix line printer. *Behavior-file* is the *rnl* or *spice* output file, *directive-file* is a "specification file" for the plot, and *plot-file* is an output file to contain the plot suitable for printing on the Printronix line printer.

The use of *mtp* involves the following steps:

1. Generate a behavior file.

If you are using *rnl*, the directive

**openplot "behavior-file"**

will cause the changes to all traced nodes to be written to *behavior-file* in addition to being written to the terminal. Quotes are necessary if the file name has any punctuation in it.

The RNL directive

**closeplot**

will terminate the behavior file. If the entire *rnl* session is to be recorded *closeplot* is not required, as the file will be terminated when *rnl* exits.

If you are using *spice*, a behavior file may be specified as the third positional parameter of the *spice* command. Behavior records will be put on this file for all nodes specified on the *Spice PLOT* directive.

2. Generate a plot file from the behavior file using *mtp*.

The plot is sent to the Printronix printer using the Unix command

**lpr plot-file**

The contents of *behavior-file* are interpreted with the help of *directive-file*. For the basic purpose of plotting the output of *rnl* or *spice*, only a few directives need be supplied:

1. **start time**

Tells *mtp* when to start plotting. If not supplied *time* defaults to 0. Data is skipped on the behavior file until an event is found whose time is greater than or equal to the start time.

2. **stop time**

Tells *mtp* when to stop plotting. A stop value must be specified. If the stop time is greater than the time of the last event on the behavior file, the plot will be concluded with the last event.

3. **scale time**

Tells *mtp* the number of time units per inch. The default value is 1000.0. Because the time unit used by *rnl* for behavior file output is 1.0 nanosecond, this value will produce plots of *rnl* output having a scale of 1.0 microsecond per inch.

4. **logical signal**

This is used primarily for plotting *rnl* output. To select signals A, B and C for plotting in logical format the directives would be

**logical A**

**logical B**

**logical C**



### 5. analog signal heights

Analog format is required when dealing with *spice* output because *spice* produces floating point values rather than logic levels. The height in inches of each trace must be specified. To select node voltages for nodes 1, 2 and 3 for plotting in analog format the necessary directives might be

```
analog V(1) 0.5
analog V(2) 0.5
analog V(3) 0.5
```

The order of selection directives in the file determines the order of the traces on the plot. The first signal selected is plotted closest to the time axis. A maximum of 20 signals may be selected on a given plot.

Spaces are used to separate the fields of a directive line. Blank lines or lines starting with # are ignored. Directives are case insensitive except for signal names.

### EXAMPLE

The following example uses *mup* to plot the behavior of a 10 bit counter, *cntr10.net*, shown here in netlist format:

```
; net file for 10-bit counter

; half adder made from gates
(macro half_adder (a b s c)
  (local h1 h2 h3)
  (nand (h1 2 16) a b)
  (nand (h2 2 16) a h1)
  (nand (h3 2 16) b h1)
  (nand (s 2 16) h2 h3)
  (invert c h1)
)

; one cell of a counter
(macro cell (in out Cin Cout)
  (local c1 c2 c3)
  (invert c1 in)
  (trans phil c1 c2)
  (invert c3 c2)
  (half_adder c3 Cin out Cout)
  (trans phi2 out in)
)

; declare global node names
(node count c in out phil phi2)

; carry-in to first significant bit controls counting action
(connect count c.0)

; generate the counter
(repeat i 1 10
  (capacitance out.i 1.234)
  (cell in.i out.i c.(1- i) c.i)
)
```

The rnl control file, cntr10.l, is as follows:

**; RNL initialization file for 10 bit ripple-carry counter**

(load "uwstd.l")

(load "uwsim.l")

(read-network "cntr10")

**; (setq report-form nil) This turns off the report generator**

(setq incr 1000)

**; bind symbols to node names**

(chflag '(phi1 phi2 out.10 out.9 out.8 out.7 out.6  
out.5 out.4 out.3 out.2 out.1))

(defun init (dummy)

(l '(count in.1 in.2 in.3 in.4 in.5  
in.6 in.7 in.8 in.9 in.10))

(l '(phi2))

(h '(phi1))

(step incr)

(l '(phi1))

(step incr)

(x '(in.1 in.2 in.3 in.4 in.5  
in.6 in.7 in.8 in.9 in.10))

(h '(phi2))

(step incr)

(l '(phi2))

(step incr)

(h '(count))

(wr-report)

'done

)

(defvec '(bit bout out.10 out.9 out.8 out.7 out.6  
out.5 out.4 out.3 out.2 out.1))

(defvec '(dec dout out.10 out.9 out.8 out.7 out.6  
out.5 out.4 out.3 out.2 out.1))

```
(def-report '("10 bit counter current state" newline " "
count (vec bout) (vec dout)))
```

Generate the behavior for the counter using *rnI*

```
netlist cntr10.net cntr10.sim
presim cntr10.sim cntr10
rnI cntr10I

init                                # initialize the counter

openplot "cntr10.evl"              # open the behavior file
                                    # (.evl stands for event list)

c 30                                # run 30 clocks

exit                                # exit rnI
```

Generate the plot.

```
mtp cntr10.evl cntr10.mtp cntr10.plt

lpr cntr10.plt
```

The file *cntr10.mtp* could contain the following:

```
start    0.0
stop     20000.0
scale    1000.0
logical  phi1
logical  phi2
logical  out.1
logical  out.2
logical  out.3
logical  out.4
logical  out.5
```

The *start* and *scale* directives are not necessary but are included for the purpose of illustration. Although not required, these directives typically precede the signal selection directives in the file.

When *mtp* runs it lists the contents of the directive file on the terminal and reports progress with the following messages:

```
Previous output cntr10.plt removed
Select and preprocess input data
Sort preprocessed events
Generate the plot
Rasterize for the Printronix
mtp complete, plot file is cntr10.plt
```

The "Rasterize for the Printronix" message marks the beginning of the longest step in the process which typically takes about a minute under moderate system loads.

*Mtp* creates scratch files named *fort.1*, *fort.2*, *fort.3*, *fort.4*, and *fort.7*. If any of these files are present when *mtp* is invoked it will exit with an error message. This can happen if *mtp* is aborted before having time to clean up the scratch files. If this happens the scratch files can be cleaned up with the Unix command

```
rm fort.[12347]
```

**SEE ALSO**

*rnl(1.vlsi)* *spice(1.vlsi)*,

*User's Guide to RNL VLSI Design Tools Reference Manual*, UW/NW VLSI Consortium, University of Washington, (Christopher Terman, MIT Laboratory for Computer Science).

*SPICE User's Guide*, VLSI Design Tools Reference Manual, UW/NW VLSI Consortium, University of Washington, (A. Vladimirescu *et al.*, 15 Oct. 1980)

**AUTHOR**

William Beckett (UW)

**NAME**

**netlist** - a simple network description language for VLSI circuits

**SYNOPSIS**

**netlist** *infile* [*outfile*] [-*e*] [-*tech*] [-*units*] [-*n*] [-*d n,m*] [-*e n,m*] [-*i n,m*] [-*l n,m*] [-*p n,m*]

**DESCRIPTION**

*Netlist* requires an input file with any/all extensions on the command line. An optional output file can be specified. Additional options are described below;

- e* Uses old input format. Size specifications are taken to be length/width rather than width/length.
- tech* Uses *tech* in the technology portion of the units/tech line at the beginning of the simulation file produced (Default is *nmos*).
- units* Sets the number of centi-microns per lambda to *units* (Default is 250).
- n* Uses number *n* as initializer for internal node names; useful when you want to merge the results of separate *netlist* runs.
- d n, n* Sets the default width to *n* and length to *m* for depletion devices. The defaults are *n*=8 and *m*=2.
- e n, n* Similar to -*d* except for enhancement devices. The defaults are *n*=2 and *m*=2.
- i n, n* Similar to -*d* except for intrinsic devices. The defaults are *n*=2 and *m*=2.
- l n, n* Similar to -*d* except for low-power devices. The defaults are *n*=2 and *m*=2.
- p n, n* Similar to -*d* except for p-channel devices. The defaults are *n*=2 and *m*=2.

In addition, if node alias records (= node1 node2 ...) are declared using "connect" (See *netlist* reference documents) they appear in a file with the name "basename.al". The *basename* is the input file name minus its last extension.

*Netlist* is a macro-based language for describing networks of sized transistors. Names in *netlist* refer to nodes, which presumably get interconnected by the user through transistors. Macros for describing transistors can be found in the *NETLIST User's Guide*. In addition to transistor macros *netlist* provides macros that allow the user to set node capacitance, specific node delays (in tenths of nanoseconds), and transistor threshold voltages. The user may also define his own macros.

The *load* command uses the environment variable *RNLPATH* (default *..\$UW\_VLSI\_TOOLS/lib/rnl*). See the *NETLIST User's Guide* for details.

**SEE ALSO**

*presim*(1.vlsi), *rnl*(1.vlsi),

*NETLIST User's Guide*, VLSI Design Tools Reference Manual, UW/NW VLSI Consortium, University of Washington,

**AUTHOR**

Christopher Terman (MIT)

**NAME**

**penplot** - display db (PLAP) or CIF files on a pen plotter

**SYNOPSIS**

**penplot** [*options*]

**DESCRIPTION**

*Penplot* is a program that plots db database files or standard CIF files on an x-y pen plotter. The current plotter options include the HP7221 and HP7580.

- db (default) Specifies that the input is from db database files (.att and .sym files);
- cif Specifies that the input is from a standard CIF file (.cif file);
- 7221 (default) Specifies that the plotter is the HP7221 x-y pen plotter;
- 7580 Specifies that the plotter is the HP7580 x-y pen plotter.

*Penplot* first prompts the user for the technology to be used. Currently they are *ames*, *lasec*, and *cmospw*. It then prompts for a data file name. This name must be of the form *dblibname/baseName* where *dblibname/baseName.cif* exists when using the -cif option, or *dblibname/baseName.att* and *dblibname/baseName.sym* exist when using the -db option. *Dblibname* is a pathname to the PLAP-generated database files.

The program can be run inside the library directory is desired.

The *penplot* commands are:

- n** Selects the symbol for subsequent plotting. The program prompts for the symbol name.
- e** Selects the symbol(s) to exclude from the next plot. The program prompts for a list of symbol names, one at a time, the last of which must be "end". Typing an "m" after a symbol name causes only the minimum bounding box of the named symbol to be plotted.
- l** Selects the layer(s) to be displayed in the next plot. The currently available layers are listed and the program prompts for a list of layers, separated by spaces and terminated by a carriage return.
- s** Lists the symbol names, including bounding boxes, in the current datafile.
- g** Sets the new grid size to some number of lambda. The program prompts for the number of lambda per grid.
- w** Changes the window definition. The current window coordinates are listed and the program prompts for new lower-left and upper-right coordinates.
- 4** Selects 4-pen plotting capability. Default is 8-pen plotting capability. To return to 8-pen mode the program must be restarted.
- 9** Toggles plotting of labels on or off.
- m** Shows the current selections in effect.
- p** Plots the current symbol with current selections in effect.
- ?** Prints the list of possible commands.
- q** Quits the program.

Roundflashes are approximated by 12 sided polygons.

The programs use the technology files to determine what color to plot each of the CIF layers. The following table shows the colors available and what pen stalls they must be in. Note that the spelling of the colors **MUST** match the spelling in the technology files. The **PLAPPATH** environment variable must contain a list of directories (in **PATH** format) in which to search

for technology files.

Stall	Color
1	black
2	blue
3	green
4	red
5	yellow, gold
6	brown
7	orange, lime-green
8	purple, violet, turquoise

Any colors not appearing in the table will be plotted as black. Furthermore, for the four pen plotters, only black, blue, green, and red are recognized with all others plotted as black.

#### FILES

*dblibname/basename.att*  
*dblibname/basename.sym*  
*dblibname/basename.cif*  
*technology.cmp*  
*technology.tec*

#### SEE ALSO

*plap(1.vlsi)*, *ca2db(1.vlsi)*, *cifplot(CAD1)*, *tec(5.vlsi)*

#### AUTHOR

(Boeing Aerospace)

#### BUGS

The tilde character '~' in pathnames always gets expanded into '/users'. Don't use it.

Default values for unselected items are not given, either by the *m* command or by the command for altering those selections.

The use of *end* to terminate the exclusion list precludes the use of any symbol named *end*.

If 4-pen plotting capability is chosen, the program must be restarted to return to 8-pen plotting capability.

The plot interrupt capability does not work when using the HP7221 four pen plotters! For some reason the enquire/acknowledge handshaking used with those plotters screws up the interrupt processing.

**NAME**

**plap** - compile, link, and run a plap layout of a VLSI circuit

**SYNOPSIS**

**plap** [-options] *basename*

**DESCRIPTION**

*Plap* will compile a PLAP IC design file in *basename.p*, link it to the appropriate artwork modules, and run the program. Template files (*technology.psh*, where *technology* is one of *nmos*, *isocmos*, *cmospw*, or *i2l*) are available in *\$UW\_VLSI\_TOOLS/lib/psh*. (*UW\_VLSI\_TOOLS* is an environment variable set to the directory that contains the vlsi tools.) Normally a designer copies the appropriate template file, renames it *basename.p*, and inserts PLAP code into it.

*options* include:

**-nocrun**

Compile and link only. Do not run.

**-nocompile**

Run only. Do not compile or link.

**-nmos**

**-isocmos**

**-cmospw**

**-i2l**

Design is being done in the respective technology, either *nmos*, *isocmos*, *cmospw*, or *i2l*.

**-sym**

Database used is db database (default).

**-ca**

Database used is Caesar database.

The program creates a number of new files:

*basename.out*

an executable file is placed in the current working directory;

*basename.ca*

a Caesar format file created when the **-ca** is used;

*symbolname.att*

an attribute file for each symbol *symbolname* defined is placed in the last library declared in the design file when the default db database format is used. If no libraries are declared, a subdirectory of the current working directory named *lib* is created (if necessary) and the attribute file is placed there;

*symbolname.sym*

a modified CIF file for each symbol *symbolname* defined is placed in the last library declared in the design file. As in the case of the .att file, the .sym file is created when the default db database format is used. If no libraries are declared, a subdirectory of the current working directory named *lib* is created (if necessary) and the modified CIF file is placed there;

*logname.log*

a log file, which is a copy of the output to the terminal, where *logname* is the string given in the PLAP *initialize* command (usually this is *basename*);



The following is a summary of commands available in a PLAP program:

#### *Artwork Primitives*

```
layer( 'layerName' );
box( x1, y1, x2, y2 );
lbox( 'layerName', x1, y1, x2, y2 );
wire( width, xStart, yStart ); wirePath
lwire( 'layerName', width, xStart, yStart ); wirePath
polygon( xStart, yStart ); polyPath [polyclose;]
```

#### *Path Primitives*

```
xy( xNext, yNext );
dxy( xDelta, yDelta );
x( xNext );
y( yNext );
dx( xDelta );
dy( yDelta );

w( newWidth );
```

#### *Symbol Definition and Call Statements*

```
define( 'symbolName' );
enddef;
draw( 'symbolName', x, y );
drawmx( 'symbolName', x, y );
drawmy( 'symbolName', x, y );
drawrot( 'symbolName', x, y, vx, vy );
```

#### *Functions and Annotation Statements*

```
lastX;
lastY;
plottext( 'textString', x, y, centered );
nodeLabel( 'textString', x, y, 'layerName' );
```

#### *Boolean Switch Variables*

```
checkswitch
manhattan
halfcheck
```

#### *Macros*

```
downroute( nLines, xStartArray, yStart, xEndArray,
           yEnd, width, spacing );
alpha( 'textString', x, y, scale );
```

#### *Symbol Access Statements*

```
library( ' libraryName' );
withsym( ' symbolName' );
fa( ' attributeString' );
setatt( ' attributeString', value );
```

**nMOS Macros**

```
z( 'newLayerName' );
rb( x, y );
gb( x, y );
```

**ISOCMOS Macros**

```
mp( x, y );
md( x, y );
be( x, y );
bm( x, y );
bs( x, y );
bw( x, y );
op( x, y );
on( x, y );
```

**CMOSPW Macros**

```
gb( x, y );
rb( x, y );
```

**FILES**

```
$UW_VLSI_TOOLS/lib/psh/*.psh
symbolname.att
symbolname.sym
symbolname.ca
basename.out
logname.log
```

**SEE ALSO**

*PLAP User's Guide, Standard Cell Library Guide, VLSI Design Tools Reference Manual, UW/NW VLSI Consortium, University of Washington.*

**AUTHOR**

(Boeing Aerospace)

**MODIFICATIONS**

(UW/NW VLSI Consortium, University of Washington)

**BUGS**

All occurrences of the string "**\$UW\_VLSI\_TOOLS**" in *basename.p* are replaced with the current value of environment variable **UW\_VLSI\_TOOLS**. However, if *basename.p* includes any files (via `#include`), it is the user's responsibility to make sure "**\$UW\_VLSI\_TOOLS**" is properly replaced in the included files.

**NAME**

**presim** - a netlist preprocessor for the **rn1** VLSI circuit simulator

**SYNOPSIS**

**presim** *infile outfile [configfile] [-g] [-cfile,min] [-tfile,min] [presist,voltage]*

**DESCRIPTION**

*Presim* converts the *.sim* file into a binary file to be used by *rn1*.

The parameters and options are as follows:

- infile*        A net list file that must include any/all extensions;
- outfile*       An output filename must be specified on the command line;
- configfile*   (optional) A file to set lambda and RC parameters for nodes and transistors in the netlist (see the *presim* user's guide for descriptions of the parameters and syntax).
- g**            Suppresses the sum-of-products formation. This may be desired if you think sum-of-products is formed wrong otherwise the advantages of the transistor and node reduction make this option unattractive.
- cfile, min**   Writes a list of node names and capacitances to the specified *file*. Only capacitances larger than *min* will be included.
- tfile, min**   Writes a list of transistors and RC values to the specified *file* -- there are two entries for each transistor. The R's come from the size of the transistor, C's from the source/drain capacitance. Only RC values larger than *min* will be included.
- presist, voltage**   Provides a worse-case estimate of the circuit power consumption by assuming that all the pullups (DEP or LOWP devices with drain=*Vdd*) are all on simultaneously. *Voltage* specifies the supply voltage,

*Presim* also attempts to open the file *basename.al*, where *basename* is defined as the input file name minus its last extension. It is non-fatal for this file to be absent. This allows *.al* files produced by Berkeley tools to be picked up easily. *Netlist* also creates this file for any (connect ...) done in the net description.

**SEE ALSO**

*netlist(1.vlsi)*, *rn1(1.vlsi)*, *simfile(5.vlsi)*

*PRESIM User's Guide*, VLSI Design Tools Reference Manual, UW/NW VLSI Consortium, University of Washington, (Christopher Terman, MIT Laboratory for Computer Science).

*RNL User's Guide*, VLSI Design Tools Reference Manual, UW/NW VLSI Consortium, University of Washington, (Christopher Terman, MIT Laboratory for Computer Science).

**AUTHOR**

Christopher Terman (MIT)

**NAME**

presto - combinational logic minimization program

**SYNOPSIS**

presto

**DESCRIPTION**

*Presto* is an efficient combinational logic minimization program. This program not only reduces the number of product terms, increases the number of don't care inputs, but also reduces the number of the output connections. Therefore, this program is very useful to pla designers.

An example of the typical input file is as follows:

```
i 4
o 2
j
p 4
10x1 11
000x 1x
1111 01
0101 10
e
```

The integer after "i" is the number of input variables. The integer after "o" is the number of output variables. The integer after "p" is the number of input product terms. "j" is optional for input listing. There is another option "d" for intermediate results.

In the input part, 1 means logic level 1, 0 means logic level 0, x(or -) means don't care. In the output part, 1 means that the term is connected to the output, 0 means that this term is not connected to the output, and x(or -) means that the output doesn't care whether this term is connected or not. "e" means the end of the input file. When there is a format error in the input file, the program will give the message: "INPUT FORMAT ERROR!" and abort the job.

**AUTHOR**

Sheng Fang

**NAME**

**pspice** - prepare an input file for the Spice circuit simulator

**SYNOPSIS**

**pspice** [-rm] [-nos2s] [-d *defsfile*] [-m *modelfile*] [-e *expfile*] *basename*

**DESCRIPTION**

*Pspice* is a shell script for preparing Spice input from information from several sources. *Pspice* runs *sim2spice* to convert from a *basename.sim* format circuit description to a Spice-compatible description and modifies the *sim2spice* node label translation table to be acceptable Spice comments. It then runs *spcpp* to translate a pseudo-Spice formatted file that contains symbolic node labels to a Spice-acceptable file. Finally, *pspice* concatenates the circuit description file, the translation table, a file of untranslated Spice input, and the translated Spice input into a single file named *basename.spcin*. This file is usually an acceptable Spice input file. The optional parameters can be used to cause parts of this process to be skipped.

The options and parameters are:

- nos2s** Suppresses the execution of the *sim2spice* step.
- rm** Indicates that the files created in intermediate steps are to be deleted.
- d *defsfile*** Specifies a file to be used as a *sim2spice* definitions file.
- m *modelfile*** Specifies a file that contains Spice input that is to be included (untranslated) in the final output. It is intended that *modelfile* name a file containing Spice MODEL cards as well as other Spice commands that are independent of the particular circuit being modeled.
- e *expfile*** Specifies a file that contains pseudo-input for Spice. *Spcpp* will interpret strings in *expfile* that are bracketed by '<' and '>' as node names to be translated into *spice* node numbers using the translation table (*basename.names*) created by *sim2spice*. Lines containing bracketed tokens are converted into Spice comments. It is intended that *expfile* contain Spice commands that describe the experiment to be simulated on the circuit. The ability to use mnemonic node names makes the preparation of Spice input much easier and it means that the description of the experiment need only be specified once, even if the circuit is modified and reextracted. If *expfile* is not specified then *spcpp* is not executed.
- basename*** Specifies the base name for the files describing the circuit. If *sim2spice* is run then a file named *basename.sim* must exist. If *sim2spice* is not run then the files *basename.names* and *basename.spice* must exist.

**FILES**

- basename.sim* circuit description input to *sim2spice*
- defsfile* optional *sim2spice* defs input
- basename.names* modified *sim2spice* translation table output. This is read by *spcpp* (\*)
- basename.spice* *sim2spice* output Spice format circuit element definitions (\*)
- modelfile* optional Spice MODEL commands to be included in *basename.spcin*
- expfile* input to *spcpp* containing pseudo-spice commands describing the experiment to be simulated
- basename.spex* translated output from *spcpp* (\*)
- basename.spcin* The Spice input deck created by concatenating *basename.spice*, *basename.names*, *modelfile*, and *basename.spex*

Note: Files marked (\*) are deleted by the -rm option.

**SEE ALSO**

*sim2spice*(1.vlsi), *spcpp*(1.vlsi)  
*spice*(1.vlsi)

mextra(1.vlsi), cifplot(1.vlsi)

**AUTHOR**

Robert Fowler (UW/NW VLSI Consortium, University of Washington)

**DIAGNOSTICS**

The error messages are intended to be self explanatory. Note that *sim2spice* and *spcpp* produce their own error messages.

**BUGS**

The command line is long enough to tempt a user to call *pspice* from yet another shell script. A better way to do this is to set up an alias for *pspice* with the commonly used options already set.

**NAME**

*rnl*, *nl* - timing and logic simulators for VLSI circuits

**SYNOPSIS**

*rnl* [*cmdfile*]

**DESCRIPTION**

*Rnl* (NetLisp) is a timing logic simulator for digital NMOS circuits with a lisp-like command interpreter. It has also been used with many CMOS circuits with some success. The *Rnl User's Guide* discusses some of the limitations found in simulating CMOS circuits. To use *rnl*, one needs a *.sim* file for the circuit to be simulated. This can be derived from the mask file (e.g., CIF) or developed using *netlist*, a program that processes textual schematics.

*Nl* is a version of *rnl* that implements a switch-level transistor model. It is called from within the *rnl* program. *Nl* is somewhat faster but does not correctly handle circuits that depend on the relative sizes of transistors for correct operation (ie., there are circuits for which *nl* and *rnl* will get different answers). However, *nl* does work for "Mead and Conway" type circuits (so called ratioless logic) -- it has been used with good results on almost all the circuits to come out of MIT in the last couple of years.

One must first convert the *.sim* file to a network file suitable for use by *rnl*. To do this run *presim*:

```
presim filename.sim netfile [config_params]
```

which converts the file *filename.sim* into *netfile*, a binary file for *rnl*. (see *Presim User's Guide* for information on the various configuration parameters.

The optional *cmdfile* is the file *rnl* initially reads for user input. Usually one prepares a command file that loads one or more library files containing RNL function definitions and reads in the network from *netfile*. As simulation proceeds, user defined functions developed for testing the circuit can be added to the command file. At a minimum the command file should contain the commands

```
(load "uwstd.l")
(load "uwsim.l")
(read-network "netfile")
```

When using the load command both *netlist* and *rnl* search the current directory and then any directories specified in the environment variable *RNLPATH*. The value of *RNLPATH* defaults to *\$UW\_VLSI\_TOOLS/lib/rnl*. Read-network does not use *RNLPATH*. *Netfile* must be produced by *presim*. When the end-of-file is reached in the command file, input is taken from stdin. Commands and formats to be used are given in the *RNL User's Guide*.

The top level of *rnl* is a simple loop:

- (1) read command from current input;
- (2) evaluate command, performing specified actions;
- (3) print the result and loop back to (1).

The following is a list of the objects that *rnl* knows about

*numbers*      -- signed integers. (16 bits on PDP11s, 24 bits on VAXen, 28 bits on PDP10s).  
                  -- floating point.

*strings*       sequences of characters enclosed in quotes ("). Useful as constants for file names, print statements, etc. Special characters can be introduced into the strings by using the backslash escapes:

```
\n'      newline
\r'      return
\t'      tab
```

- symbols**     `\ooo'`     ascii code "ooo" where ooo are octal digits
- symbols**     variable names. Any sequence of characters that isn't a number, string, or some special character -- starting symbols with a letter, followed by more letters, numbers, and punctuation is usually a safe bet.
- nodes**        an electrical node.
- lists**        a sequence of objects enclosed in parentheses. Standard LISP syntax applies, including dot notation. The empty list "()" is also called "nil".
- subs**        primitive, or built-in, functions (like +).

The functions are listed by application area. The areas are:

- arithmetic functions
- predicates
- list functions
- I/O functions
- miscellaneous functions
- special forms
- network/simulation functions
- functions defined in "uwsim.l"

#### SEE ALSO

*netlist(1.vlsi)*, *presim(1.vlsi)*, *simfile(5.vlsi)*

*RNL User's Guide, VLSI Design Tools Reference Manual*, UW/NW VLSI Consortium, University of Washington, (Christopher Terman, MIT Laboratory for Computer Science).

#### AUTHOR

Christopher Terman (MIT)

#### BUGS

User defined macros with the same name as a node in the net list puts *nil* into an infinite loop.



**NAME**

**simfilter** - Berkeley *.sim* <--> MIT *.sim* filter for VLSI simulators

**SYNOPSIS**

**simfilter** [-*lambda* | -*clambda* | -*llambda*] *infile* [*outfile*]

**DESCRIPTION**

*Simfilter* reformats the *.sim* file format of either Berkeley or MIT (output from *mextra* for example) into the other *.sim* format. This allows layouts extracted by *mextra* to be simulated using *rn1*. Alternatively, using the -*n* or -*e* option allows *.sim* files from *netlist* to be input for *sim2spice* or *crystal*.

"r", "C", "c", "N", *nmes* and *cmes* transistor records are accepted and reformatted. The so-called attribute records "A" in the Berkeley format and the "=" records in the MIT format are reformatted as comments.

*Simfilter* requires a *.sim* file, *infile* including any/all extensions for input. Output defaults to standard output.

Additional command line options are:

-*lambda* Outputs *.sim* file in Berkeley format with technology set to *nmes*. Lambda is given in centimicrons and defaults to 200.

-*clambda* Outputs *.sim* file in Berkeley format with technology set to *cmes-pw*. Lambda is given in centimicrons and defaults to 200.

-*llambda* Outputs *.sim* file in Berkeley format with technology set to *lscmes*. Lambda is given in centimicrons and defaults to 200.

These options would generally be used on output files from *netlist*. Spaces between -*n*, -*c* or -*l* and *lambda* are not accepted.

*outfile* (optional) An output file name can be specified on the command line.

Note: The default *lambda* values for Berkeley and MIT *.sim* files differ. *Simfilter* makes no assumptions about compatibility of these parameters when reformating Berkeley *.sim* files. This must be done during the extraction of the layout or when running *presim* on the *.sim* file for *rn1*. See the appropriate documentation for details.

**SEE ALSO**

*mextra*(1.vlsi), *netlist*(1.vlsi), *presim*(1.vlsi), *rn1*(1.vlsi)  
*cifplot*(1.vlsi),

**AUTHOR**

Rob Daasch (UW/NW VLSI Consortium, University of Washington)

**BUGS**

**NAME**

**simtools** - Introduction to simulation support tools for VLSI

**DESCRIPTION**

This document is an introduction to the tools in the UW/NW-VLSI Consortium collection that are used in connection with the simulation of VLSI circuits. These fall into two categories. The first category is the set of simulators and circuit checkers. The second is a set of filters and utilities that are useful for preparing inputs to the simulators and for converting from one file format to another.

We note here that the circuit extractor *mextra* is used for obtaining *.sim* files from layouts.

The simulators and checkers are:

- spice2g6***     The well known device level circuit simulator, with minor mods to output an additional file that allows multiple time series plots to be made.
- rnl***           An event driven "timing" simulator. It uses logic levels and a simplified circuit model to estimate timing delays through digital circuits. It also has a mode that allows it to be used as a switch (gate) level simulator.
- esim***          An NMOS gate level simulator. It uses logic levels and models transistors as perfect switches.
- crystal***      A static timing verifier. It uses a simplified circuit model to estimate the worst case delay through a circuit.
- erc***           Checks circuit description of an NMOS design for consistent electrical properties.

The filters and utilities that are included in the distribution are:

- forfor***       Converts a file using FORTRAN style carriage control to one using ASCII control characters.
- netlist***       Generates circuit descriptions (net lists). The output is a *.sim* file.
- presim***       Converts a *.sim* file into the binary format required by *rnl*. In the process *presim* simplifies the circuit by identifying gates in the circuit.
- pspice***       Runs *sim2spice* and *spcpp*. In addition to running those programs it concatenates various files so as to create a complete Spice input deck. This considerably reduces the effort of simulating variations on a particular circuit.
- sim2spice***     Reads a *.sim* file containing a description of a circuit. It writes a *.names* file and a *.spice* file. The former contains a translation file from node names in the *.sim* file to the Spice node numbers. The *.spice* file contains a description of the devices in the circuit in a form acceptable to Spice.
- spcpp***       Facilitates the writing of Spice input by allowing the user to refer circuit nodes using mnemonic node labels rather than Spice node numbers.
- spice***       A *csh* script for running *spice2g6*.
- mtp***           Displays, as multiple time series plots on a Printronix printer, the simulation output from either *spice* or *rnl*.
- simfilter***    Translates between Berkeley style *.sim* files and MIT style *.sim* files.

**NAME**

**spcpp** - Spice (circuit simulator) input pre-processor

**SYNOPSIS**

**spcpp** [-c] [-sn] [-d lr] [-t *iname*] [-e *oname*] *iname*

**DESCRIPTION**

*Spcpp* is a program that translates bracketed text tokens in an input file into other text strings. It is intended to allow users of *spice* to prepare their simulation input using mnemonic node names rather than the numeric node numbers required by Spice commands. The program has two major modes of operation. If the user does not specify a file that contains a translation table, then *spcpp* builds a translation table itself numbering the tokens from zero as it encounters them. Alternatively, the user can specify the name of a file containing a translation table to be used. In particular, the *.names* file created by *sim2spice* is usable as a translation table file.

The options and parameters are:

- c Indicates that the first non-whitespace word of each line of the translation table file should be skipped over. This is useful if your translation table has an asterisk (\*) in column 1 of each line to allow it to be read by *spice* as comments.
- sn Indicates that *n* lines at the beginning of the translation table file should be skipped over. If no number is specified then only the first line of the file is skipped.
- d lr Redefines the token delimiters to be 'l' and 'r' respectively. The default delimiters are '<' and '>'.
- t *iname* Specifies a file that contains a translation table (default is to build a translation table as described above). Each line of this file should have at least two non-whitespace words on it. If the -c option is specified then the first word on each line is ignored. The next word is interpreted as a string to be translated and following one is interpreted as the target string into which it is translated. Any subsequent words on the line are ignored. For Spice input preparation the target string should be a numeral. The -s option allows the file to be prefaced by one or more lines that *spcpp* will ignore.
- e *oname* Specifies a file into which the output is to be written. If this option is not used then the output is written to *lroot.spex* where *lroot* is obtained by stripping away any tags from *iname*.
- iname* Specifies the name of the file to process.

A bracketed token is defined to be a left delimiter character, zero or more spaces, a word (the token) not containing either right or left delimiters, zero or more spaces, and a right delimiter character. Unmatched delimiter characters are not allowed in any context. Bracketed tokens are not allowed to span lines. Tokens and the strings that they translate into are limited to be at most 40 characters each.

Any line that contains no bracketed tokens is simply copied from the input to the output. If a line does contain a bracketed token then the input line is written into the output a Spice comment line. An output line follows immediately. If the line is valid, then the output line has the untranslated parts immediately below the corresponding parts of the commented input line with the target strings substituted for the bracketed tokens. If an error is detected, then the output line has a caret (^) immediately below the point at which the first error is detected. An error message line then follows. Since the scanning of the line is abandoned there may be subsequent undetected errors in the remaining part of the line.

**Example:**

If the following lines are contained in the translation table file:

```
Vdd 1
Input 55
Output 107
foo 23
bar 45
```

then *spcpp* will, upon seeing the lines:

```
.plot trans v(<Input>) v(<Output>), i(<Vdd>)
+ v(<foo>), v(<bar>)
```

will output the lines:

```
* .plot trans v(<Input>) v(<Output>) v(<Vdd>)
  .plot trans v(55) v(107) v(1)
* + v(<foo>), v(<bar>)
  + v(23), v(45)
```

Note that *spcpp* correctly handles Spice continuation cards.

Note also that the substitution process is not recursive. That is, once a token has been translated, the translated string is not rescanned.

The usefulness of *spcpp* for simulating a circuit extracted from a layout depends upon the user being able to ensure that his mnemonic node labels will be retained through the extraction process. The *mextra* and *sim2spice* manual entries will help with this.

*Pspice* is a shell script that runs *sim2spice* and *spcpp* and concatenates several files is useful for preparing Spice inputs from .sim files.

**FILES**

```
iname
iroot.spcx
oname
tname
```

**SEE ALSO**

*mextra*(1.vlsi), *pspice*(1.vlsi), *sim2spice*(1.vlsi), *simtools*(1.vlsi), *spice*(1.vlsi),

*SPICE User's Guide*, VLSI Design Tools Reference Manual, UW/NW VLSI Consortium, University of Washington, (*SPICE Version 2G6 User's Guide*, A. Vladimirescu et al., 15 October 1980)

**AUTHOR**

Robert Fowler (UW/NW VLSI Consortium, University of Washington)

**DIAGNOSTICS**

The error messages are intended to be self explanatory. If *spcpp* encounters a syntax error on a line then it suspends processing on that line and writes it as a Spice comment to the output file. It then writes a line containing a caret (^) under the character at which scanning failed and finally, a line containing an error message. It then goes on to process the remaining lines of the file. If errors have been encountered then at the end of the output file *spcpp* writes messages to the effect that errors have been encountered and exits with status 1. The error

messages written to the output file begin with dollar signs. In addition, some number of messages are directed towards the standard error output.

**BUGS**

The target strings are not checked to see whether they are valid numerals or not. This can be regarded as either a bug or a feature.

The target string must fit into the space from the left to right token delimiter inclusive. This is normally not a problem since most node numbers will be small integers and the available space will be at least three characters. This was done so that the input lines and the translated outputs would line up vertically.

**NAME**

**spice** - circuit simulator

**SYNOPSIS**

**spice** *infile outfile [mupfile]*

**DESCRIPTION**

*Spice* reads a circuit description from *infile*. Output is written to *outfile*, and error messages to standard error. An optional output file, *mupfile*, can be used by *mup* to obtain a multiple time series plot on a Printronix.

*Spice* calls *spice2g6*, a general-purpose circuit simulation program for nonlinear DC, nonlinear transient, and linear AC analyses. Circuits may contain resistors, capacitors, inductors, mutual inductors, independent voltage and current sources, four types of dependent sources, transmission lines, and the four most common semiconductor devices: diodes, BJTs, JFETs, and MOSFETs.

*Spice2g6* has built-in models for the semiconductor devices, and the user need specify only the pertinent model parameter values. The model for the BJT is based on the integral charge model of Gummel and Poon; however, if the Gummel-Poon parameters are not specified, the model reduces to the simpler Ebers-Moll model. In either case, charge storage effects, ohmic resistances, and a current-dependent output conductance may be included. The diode model can be used for either junction diodes or Schottky barrier diodes. The JFET model is based on the FET model of Shichman and Hodges. Three MOSFET models are implemented; MOS1 is described by a square-law I-V characteristic, MOS2 is an analytical model while MOS3 is a semi-empirical model. Both MOS2 and MOS3 include second-order effects such as channel length modulation, subthreshold conduction, scattering limited velocity saturation, small size effects and charge-controlled capacitances.

To build a Spice input file for your circuit from *mextra* output run *sim2spice* or *pspice*.

**SEE ALSO**

*mextra*(1.vlsi)

*mtp*(1.vlsi), *sim2spice*(1.vlsi), *pspice*(1.vlsi), *spcpp*(1.vlsi)

*SPICE User's Guide, VLSI Design Tools Reference Manual*, UW/NW VLSI Consortium, University of Washington, (*SPICE Version 2G6 User's Guide*, A. Vladimirescu *et al.*, 15 October 1980).

*Program Reference for Spice2*, E. Cohen, ERL Memo. ERL-M592, Electronics Research Laboratory, University of California, Berkeley, June 1976.

*SPICE2: A Computer Program to Simulate Semiconductor Circuits*, L.W. Nagel, ERL Memo. ERL-M520, Electronics Research Laboratory, University of California, Berkeley, May 1975.

*The Simulation of MOS Integrated Circuit Using SPICE2* A. Vladimirescu and Sally Liu, UCB/ERL M80/7, University of California, Berkeley, February 1980.

**AUTHOR**

(UCB)

**BUGS**

MOSFET Model, Level=2 does not work, due to a charge conservation problem (it grows).

**NAME**

**vic** - view an integrated circuit layout.

**SYNOPSIS**

**vic** [*options*] *symbolname*

**DESCRIPTION**

*Vic* is an interactive graphics display program for integrated circuits that is technology independent and has a built-in hardcopy feature. It understands layouts in db and Caesar data base format. It currently displays only on the GP19 graphics terminals, but it can produce a hardcopy on a number of devices.

The *options* are as follows:

**-t technology**

Supported values of *technology* are *l2l*, *names*, *lscmes*, and *cncspw*. Default is *lscmes*.

**-h plotter**

Supported values for *plotter* are *HP7221CT*, *HP7221AB*, *HP7500*, and *Tek4662\_31*. Default is *HP7221CT*.

**-g graphics**

Currently there is only one valid value for *graphics*, *GP19*.

**-9**

echoes the names of the symbols as they are read in.

**-f format**

Choices for *format* of symbol to be read are *sym* (db files) or *ca* (Caesar files). Default is *sym*.

**COMMANDS**

For all the commands, only the portion enclosed in parentheses need be typed and a list of the possible parameters for each command (if any) are shown after the command in the menu.

**(lay)ers <list>**

list the layers to be plotted. The list consists of layer names separated by spaces, or the entire list may be preceded by a "+". In the latter case, the given layers are added to the plot ALREADY on the screen (it should be pointed out that a space must follow immediately after the "+", followed by the additional layers). Upper/lower case is not significant. A null list sets all layers to be plotted. Default is all layers.

**(n)esting levels <number>**

set the number of levels in the symbol's hierarchy to be plotted. Any symbol at a level greater than this will show up only as a bounding box with its symbol name in the lower left corner. The current symbol is at level 1, its children are at level 2, and so on. Default is 1.

**(w)indow**

window in on the plot. Use the graphics cursor to move to the desired lower left corner of the window and hit the space bar. Then move to the upper right corner and do the same.

**(h)ardcopy <device>**

produce a hardcopy of exactly what is shown on the terminal screen on either the pen plotter, versatec plotter, or printtronix printer. A grid may also be placed over the hardcopy by specifying anything greater than zero when the program prompts for grid size. For versatec and printtronix hardcopies, the program calls the *cp/plot(1.vni)* program with the current plotting options. Default is *penplot*. For the *penplot* option to work, the user's terminal must communicate with the host through the plotter, in order that the plotter may intercept the plotting commands.

**(lab)els < value>**

turn node labels on/off. Default is on.

**(p)lot** plot on the graphics terminal with the current options in effect.

**(v)iew** view the current symbol fully instantiated with all layers and node labels.

**(he)lp** show the menu.

**(q)uit** quit from the program.

**(s)ymbol < name>**

select the symbol to be plotted. Default is the highest level symbol.

#### DIAGNOSTICS

If an error occurs, a message is written to standard error and the program exits with a non-zero status.

#### FILES

*technology.tec*

*technology.cmp*

*symbol.att*

*symbol.sym*

*symbol.ca*

#### SEE ALSO

caesar(CAD1), cifplot(1.vlsi), penplot(1.vlsi), dbdi(1.vlsi), tec(5.vlsi)

#### AUTHORS

Pat Bates

Larry McMurchie

Bruce A. Yanagida



**NAME**

simfile - sim file file format

**DESCRIPTION**

The .sim files are ASCII files used by various programs to describe MOS transistor networks and their associated parameters. *Mextra* and *netlist* output .sim files; *sim2spice*, *presim* and others read the files.

Each line of a .sim file is a separate "record" whose type is determined by the first character of the line. The possible record types are:

! ...

Lines beginning with vertical bar (!) are treated as comments and ignored by programs that read .sim files.

A only exception is if this line contains the information units:, tech: and format:. Units specifies the conversion factor to centimicrons, tech declares the technology (e.g. nmos) that was used in the design and format declares the .sim file format used for the various other records.

The format specifier is required as there are currently two slightly different formats used. One is used by tools from Berkeley and the other by tools from MIT. If a format isn't declared in one of these interpreted comments the original file format used by the tool is assumed.

The only program that uses a .sim file for input that doesn't support both formats is the *erc*. This is because it is heavily dependent on the circuit extractor *mextra*.

Other than the record formats the main cause for concern are the units used in transistor sizes and capacitive loads. MIT generally uses a lambda value for transistor size and picofarad loads, UCB uses centimicron transistor sizes and femtofarad loads.

@ filename

Redirects input from the named file. When the end-of-file is reached, input reverts to the current file at the following line. Indirect files can be nested; usually there is some system dependent limit on the number of simultaneously open files which limits the depth of nesting.

The following record is format dependent;

**MIT format:**

e gate source drain length width xpos ypos [rpa] area  
 l gate source drain length width xpos ypos [rpa] area  
 d gate source drain length width xpos ypos [rpa] area  
 p gate source drain length width xpos ypos [rpa] area  
 l gate source drain length width xpos ypos [rpa] area

**UCB format:**

e gate source drain length width xpos ypos [g=att s=att d=att]  
 p gate source drain length width xpos ypos [g=att s=att d=att]  
 d gate source drain length width xpos ypos [g=att s=att d=att]  
 n gate source drain length width xpos ypos [g=att s=att d=att]

Above are the possible transistor records. The first character tells the type of the transistor:

**MIT format:**

- e n-channel enhancement
- l n-channel zero threshold (intrinsic) enhancement
- p p-channel enhancement
- d depletion
- l low-power depletion

**UCB format:**

- e n-channel enhancement
- p p-channel enhancement
- d depletion
- u unusual implant

The next three parameters are the names of the *gate*, *source*, and *drain* nodes.

The *length* and *width* can be either integers or floating-point giving the dimensions of the active transistor area.

*xpos* and *ypos* report the coordinates of the upper left hand corner of the transistor (NETLIST always specifies 0,0).

The next parameter (MIT only) is a letter specifying the shape of the transistor:

- r rectangular
- p rectangular, monotonically increasing width
- a other shapes...

*Netlist* always puts an "r" in this field.

*area* (MIT only) tells the true active area; may be different from width\*length if the network extractor used an approximation (in output from *netlist*, area always equals width\*length).

G=att, s=att and d=att (UCB only) are optional transistor attributes used by *crystal* a timing verifier.

The following record is format dependent;

**MIT format:**

*N node xpos ypos MA PA DA DP*

**UCB format:**

*N node DA DP PA PP MA PP*

Node record from the network extractor. Tells the node name, coordinates of the upper left corner of the node and various geometrical info described below.

MIT format only;

*M node xpos ypos M2A M2P MA MP PA PP DA DP PDA PDP*

This record is similar to (N) in a format (MIT only) that includes more information.

*xpos* and *ypos* give the coordinates of the upper left corner of the node and following provide geometrical information:

- M2A* area of 2nd-level metal
- M2P* perimeter of 2nd-level metal
- MA* area of 1st-level metal

*MP* perimeter of 1st-level metal

*PA* area of polysilicon

*PP* perimeter of polysilicon

*DA* area of n-diffusion

*DP* perimeter of n-diffusion

*PDA* area of p-diffusion

*PDP* perimeter of p-diffusion

*Presim* and *sim2spice* can use these numbers to compute interconnect capacitances from layout information; *netlist* never produces this type of record. Note that not all programs use all of these values for computing the lumped capacitor value. The perimeter measures are the actual total perimeters.

The following record is format dependent.

MIT format:

*c node cap*

UCB format:

*C node1 node2 cap*

Used to specify *node* capacitance in *pf. cap* can be either an integer or floating-point number. This is the type of record used by *netlist* to described user-specified capacitors. Node2 is assumed GND in the MIT format.

*= node1 node2 ...*

Used to establish a name equivalence. Indicates that two or more names refer to the same electrical node. Parameters for the specified nodes are merged; the user can subsequently use the names interchangeably.

The following are MIT records and have no UCB equivalent.

*D node low-to-high high-to-low*

Describes user-specified delays for the given node. Delays are integral numbers of RNL time units (1/10th nanosecond).

*t node low high*

User specified node threshold. Describes user-specified voltage thresholds in normalized voltage units (i.e., low and high are floating-point numbers in the range 0.0 to 1.0).

The following is a UCB record and has no MIT equivalent.

*A node attribute*

This is the node equivalent of the transistor attribute mentioned earlier and is also used by the timing verifier *crystal*.

#### SEE ALSO

*mextra*(CAD1)

*presim*(1.VLSI)

*netlist*(1.VLSI)

*sim2spice*(CAD1)

errors	o	34	255	0	extra
buried_contact	o	32	255	0	buried
LL	s	35	255	0	label

Here, *nmos* refers to *nmos.cmp*, which is the colormap to be used. The columns are defined as follows:

- 1) plap layer name;
- 2) fill type (f=filled, x=X'ed, o=outlined);
- 3) color map index (see below);
- 4) write mask (255=opaque, others control amount of transparency);
- 5) minimum wire width in lambda;
- 6) caesar layer name.

The *technology.cmp* file contains the color map used by the plotting programs. There are 256 entries (0-255) and all must be specified. Below is a portion of the default *nmos.cmp* file:

200	200	200	0	light background
220	0	120	1	red
70	250	70	2	green
160	160	0	3	
0	0	200	4	blue
115	0	155	5	
0	170	140	6	
115	115	125	7	
250	250	0	8	yellow
200	100	60		

The columns are defined as follows:

- 1) RED value (8 bits);
- 2) GREEN value (8 bits);
- 3) BLUE value (8 bits);
- 4) color map index (as referenced in *technology.tec* and *technology.tec2*);
- 5) color name.

#### SEE ALSO

*technology(5)*

**NAME**

technology - VLSI technology database files

**SYNOPSIS**

*technology.tec*  
*technology.tec2*  
*technology.cmp*  
*technology.tech*  
*technology.std*  
*technology.pale*  
*p-style.tp*  
*pat.technology*  
*cellib/technology/*  
*technology.r*  
*technology\**  
*technology.prm*

**DESCRIPTION**

Technology files used by various VLSI programs. Each file is described below giving the file name with specific technologies available, a short description of the file, the directory containing the source file, file access information, reference to file format information and a list of tools that use the file. Refer to the manual pages for the tools for the specific usage of a file.

**FILE DESCRIPTION:**

*technology.tec* - layer information and colormap name  
*technology.tec2* - PLAP-CAESAR communication information  
*technology.cmp* - colormap to be used by plotting programs

where *technology* is one of

*cmospw* - MOSIS 3 micron bulk cmos process  
*isocmos* - GTE 5 micron isocmos process  
*nmos* - MOSIS nmos process

**TOOLS USING THESE FILES:** *plap, penplot, vic*

**DEFAULT VERSIONS:** *\$UW\_VLSI\_TOOLS/lib/technology/*

**INSTALLED LOCATION:** Environment variable **PLAPPATH** is searched for these files, so the user may create these technology files in any directory specified in this search path.

**FORMAT INFORMATION:** see *tec(5)*.

**FILE DESCRIPTION:**

*technology.tech* - Caesar layer information  
*technology.std* - Caesar color map  
*technology.pale* - alternate Caesar color map (for long persistence phosphors)

where *technology* is one of

*cmos-pw* - MOSIS 3 micron bulk cmos process  
*isocmos* - GTE 5 micron isocmos process (no *isocmos.pale* file available)  
*nmos* - MOSIS nmos process

**TOOLS USING THESE FILES:** *caesar, ctf2ca, lyra, tpack*

**DEFAULT VERSIONS:** `~cad/lib/caesar/`

**INSTALLED LOCATION:** The Caesar search path is searched for *technology.tech* when executing the 'technology' long command. The first *technology.tech* file encountered is used. If no file is encountered in the Caesar search path, caesar looks in directory `~cad/lib/caesar`. A similar resolution occurs when loading a new colormap file via the *xcload* long command. The colormap file may be a *technology.std* file or a *technology.pale* file.

**FORMAT INFORMATION:** see "Editing VLSI Circuits with Caesar" by John Ousterhout.

**FILE DESCRIPTION:**

*p-style.tp* - pla style files for tpla

where *style* is one of (see tpla manual page for more detail)

Bcis	- nmos (buried contacts), 'cis' version
Btrans	- nmos (buried contacts), 'trans' version
Mcis	- nmos (butting contacts), 'cis' version
Mtrans	- nmos (butting contacts), 'trans' version
Tcis	- Similar to Bcis
Ttrans	- Similar to Btrans
isocmos	- GTE 5 micron isocmos process (floating gates)
isocmos_g	- GTE 5 micron isocmos process (gates tied to ground)
CS3cis	- MOSIS 3 micron pwell cmos, 'cis' version
CS3trans	- MOSIS 3 micron pwell cmos, 'trans' version

**TOOLS USING THIS FILE:** *tpla*

**DEFAULT VERSIONS:** `~cad/lib/tpla/`

**INSTALLED LOCATION:** The style of pla to generate is specified with the *-s* option in tpla. The directory `~cad/lib/tpla/` is searched for the file.

**FORMAT INFORMATION:** see the *tpla* manual page

**FILE DESCRIPTION:**

*pat.technology* - stiple patterns for each layer for cifplot

where *technology* is one of

cmospw	- MOSIS 3 micron bulk cmos process
isocmos	- GTE 5 micron isocmos process
nmos	- MOSIS nmos process

**TOOLS USING THIS FILE:** *cifplot*

**DEFAULT VERSIONS:** `~cad/lib/`

**INSTALLED LOCATION:** The user may specify the file via the *-P* cifplot option or through use of the *.cadrc* file (see *cadrc* man page). The default is nmos. User-created pattern files must be referred to with full path name using the *-P* cifplot option or the *.cadrc* file.

**FORMAT INFORMATION:** see the *cifplot* manual page

**FILE DESCRIPTION:**

*celllib/technology/* - standard cell library

where *technology* is one of

<i>cmospw</i>	- MOSIS 3 micron bulk cmos process
<i>isocmos</i>	- GTE 5 micron isocmos process
<i>nmos</i>	- MOSIS nmos process

**TOOLS USING THESE DIRECTORIES:** *plap, caesar*

**DEFAULT VERSIONS:** \$UW\_VLSI\_TOOLS/lib/celllib/.

**INSTALLED LOCATION:** PLAP: the user must include the library location in plap source code. Caesar: the user's standard cell library must have the directory name somewhere in the search path.

**FORMAT INFORMATION:****FILE DESCRIPTION:**

*technology.x* - lyra rule set lisp source

*technology\** - compiled lyra rule set

where *technology* is one of

<i>cmospw3</i>	- MOSIS 3 micron bulk cmos process
<i>isocmos</i>	- GTE 5 micron isocmos process
<i>nmosBERK</i>	- MOSIS nmos process (buried contacts)
<i>nmosMC</i>	- Mead & Conway rules (butting contacts)

**TOOLS USING THESE FILES:** *lyra, caesar*

**DEFAULT VERSIONS:** The source and compiled versions of these files are contained in the default directory *~cad/lib/lyra*

**INSTALLED LOCATION:** User-created, compiled rule sets must be in *~cad/lib/lyra* or the full path name of the file must be specified using the *-r* option.

**FORMAT INFORMATION:** see "Specifying Design Rules for Lyra" by Michael H. Arnold, 29 March 1983.

**FILE DESCRIPTION:**

*technology.prm* - spice parameters

where *technology* is one of

<i>cmos2</i>	- sample MOSIS bulk cmos process, Level=2
<i>gteiso2</i>	- GTE 5 micron isocmos process, Level=2
<i>nmos2</i>	- MOSIS 2 micron nmos process, Level=2
<i>nmos3</i>	- MOSIS 2 micron nmos process, Level=3

**TOOLS USING THIS FILE:** *pspice*

**DEFAULT VERSIONS:** Example versions of these files are in  
\$UW\_VLSI\_TOOLS/src/spice/params/

**INSTALLED LOCATION:** User versions of these files must be in the current directory or must have the full path name specified in the -m option of *pspice*.

**FORMAT INFORMATION:** see "SPICE User's Guide" by Vladimirescu, *et al.*

**Tool to Technology File Table**

<b>Tools</b>	<b>Generic nmos</b>	<b>GTE isocmos</b>	<b>MOSIS 3 micron cmos</b>
penplot, plap, vic	nmos.tec nmos.tec2 nmos.cmp	isocmos.tec isocmos.tec2 isocmos.cmp	cmospw.tec cmospw.tec2 cmospw.cmp
caesar, cif2ca, lyra, tpack	nmos.tech nmos.std nmos.pale	isocmos.tech isocmos.std	cmos-pw.tech cmos-pw.std cmos-pw.pale
cellib (caesar, plap)	./nmos/	./isocmos/	./cmospw/
tpla	p-Bcis.tp p-Btrans.tp p-Mcis.tp p-Mtrans.tp p-Tcis.tp p-Ttrans.tp	p-isocmos.tp p-isocmos_g.tp	p-CS3cis.tp p-CS3trans.tp
cifplot	pat.nmos	pat.isocmos	pat.cmospw
lyra	nmosMC.r nmosMC* nmosBERK.r nmosBERK*	isocmos.r isocmos*	cmos-pw3.r cmos-pw3*
pspice	nmos2.prm nmos3.prm	gteiso2.prm	cmos2.prm



**NAME**

**.cadrc** – Initialization file (Modified by UW/NW VLSI Consortium)

**DESCRIPTION**

The **.cadrc** file is an ASCII text file which is used to initialize several CAD programs. A **.cadrc** file may be in any subdirectory or the users home directory. A direct search from the current subdirectory to the home directory is done, stopping at the first **.cadrc** file found. This encourages the user to build separate **.cadrc** files for specific projects and hopefully minimize conflict between projects.

In addition to the **.cadrc** file in the user's home directory there may be a **.cadrc** file in **cad**. This file is read before the one in the user's directory and is used to tell the program where it can find various files and library programs. This allows program binaries to be transported between systems without recompiling.

The **.cadrc** file contains blocks identified by (Begin Program-Name, end) pairs. Each block contains commands listed on separate lines. Only those commands within the block identified by the current program being executed will be processed.

The first word on the line is called the *keyword*. The keyword tells the program how to interpret the line. When a program reads a keyword it doesn't understand it ignores the line. The case of the keyword is ignored. Text following a ";" is ignored and can be used to comment the **.cadrc** file. What follows is a list of **.cadrc** command lines.

**Echo** This command writes to stderr the absolute path of the **.cadrc** file being used.

**AreaToCap** *layer value*

This command is read by the cifplot circuit extractor and mextra. It is used to set up the default capacitance per unit area. *layer* can be 'metal', 'poly', 'diff', or 'poly/diff'. *value* is in atto-farads (10<sup>-18</sup> farads) per square micron. Also see the command 'perimertocap'.

**CapThreshold** *value*

This command is read by mextra. Mextra will not report any node capacitance below *value*. *value* is in femto-farads.

**Cifplot** *options*

This line allows you to select default command line options for cifplot. Call this command just as you would call cifplot from the shell but without any CIF file.

**Device** *DevCh xmax ymax resolution DumpProg*

This command sets up information about a particular plotting device. This command is used by cifplot. *DevCh* is a single character which indicates which output device. The characters 'U', 'V', and 'W' are black and white raster scan type devices. Lower case letters are for output in trapezoid format and is generally used for driving random access displays. The letter 'P' is for pen plotters. *DumpProg* is the program to actually display the plot on the device. For raster scan output the program is called with the the name of the dump file. For other type of devices the program is called so that information is piped into standard input. *xmax* and *ymax* indicated the range in device co-ordinates in the x and y direction. *resolution* is the resolution of the device in dots per inch.

**FontDir** *dirname*

*dirname* is the name of the directory in which to find font files. This keyword is recognized by cifplot.

**MachineName** *name*

*name* is the net address of the machine. (E.g. on Ernie the the command would be "machinename cvax".)

**MaxLength** *length*

*length* specifies the maximum length in feet that can be plotted. This command is recognized by *cifplot*.

**PatFile** *file*

*file* is a file of stipple patterns to be used as the default stipple. This command is recognized by *cifplot*.

**PerimeterToCap** *layer value*

This command is read by the *cifplot* circuit extractor and *mextra*. It is used to set up the default capacitance per unit length. *layer* can be 'metal', 'poly', 'diff', or 'poly/diff'. *value* is in atto-farads ( $10^{-18}$  farads) per micron. Also see the command 'areatocap'.

**Sim2Spl** *TransType parameter value*

This command is read by *sim2spl*. It is used to set default parameters to give to *splice*. *TransType* is a 'sim' transistor type, either 'e' or 'd'. *parameter* is one of the splice parameters: 'vt', 'kp', 'gam', 'phi', or 'lam'. *value* is the value given to that parameter.

**TmpDir** *dirname*

*dirname* is the name of a directory with a lot of free space. This directory is used to set up dump files by *cifplot*.

A brief example may clarify what is going on here.

Example:

```

Begin Sim2spice
    tech cmos-pw ; technology set to cmos-pw
    set vdd 1    ; identify new nodes
    set gnd 0
end
begin mextra
    tech isocmos ; technology for mextra set
                  ; set to isocmos
    echo         ; cadrc used will echo to stderr
END             ; Note case insensitivity

```

**SEE ALSO**

*cifplot*(CAD)  
*mextra*(CAD)  
*sim2spl*(CAD)

**BUGS**

Not yet completely implemented or documented.

**NAME**

**cifplot** - CIF interpreter and plotter for displaying VLSI circuits

**SYNOPSIS**

**cifplot** [*options*] *file1.cif* [*file2.cif* ...]

**DESCRIPTION**

*Cifplot* takes a description in Cal-Tech Intermediate Form (CIF) and produces a plot. CIF is a low-level graphics language suitable for describing integrated circuit layouts. Although CIF can be used for other graphics applications, for ease of discussion it will be assumed that CIF is used to describe integrated circuit designs. *Cifplot* interprets any legal CIF 2.0 description including symbol renaming and Delete Definition commands. In addition, a number of local extensions have been added to CIF, including text on plots and include files. These are discussed later. Care has been taken to avoid any arbitrary restrictions on the CIF programs that can be plotted.

To get a plot call *cifplot* with the name of the CIF file to be plotted. If the CIF description is divided among several files call *cifplot* with the names of all files to be used. *Cifplot* reads the CIF description from the files in the order that they appear on the command line. Therefore the CIF *End* command should be only in the last file since *cifplot* ignores everything after the *End* command. After reading the CIF description but before plotting, *cifplot* will print a estimate of the size of the plot and then ask if it should continue to produce a plot. Type *y* to proceed and *n* to abort. A typical run might look as follows:

```
% cifplot lib.cif sorter.cif
Window -5700 174000 -76500 168900
Scale: 1 micron is 0.004075 inches
The plot will be 0.610833 feet
Do you want a plot? y
```

After typing *y* *cifplot* will produce a plot on the Benson-Varian plotter.

*Cifplot* recognizes several command line options. These can be used to change the size and scale of the plot, change default plot options, and to select the output device. Several options may be selected. A dash (-) must precede each option specifier. The following is a list of options that may be included on the command line:

**-w *xmin xmax ymin ymax***

(*window*) This option specifies the window; by default the window is set to be large enough to contain the entire plot. The windowing commands lets you plot just a small section of your chip, enabling you to see it in better detail. *Xmin*, *xmax*, *ymin*, and *ymax* should be specified in CIF coordinates.

**-s *float***

(*scale*) This option sets the scale of the plot. By default the scale is set so that the window will fill the whole page. *Float* is a floating point number specifying the number of inches which represents 1 micron. A recommended size is 0.02.

**-l *layerlist***

(*layer*) Normally all layers are plotted. This option specifies which layers NOT to plot. The *layerlist* consists of the layer names separated by commas, no spaces. There are some reserved names: *allText*, *bbox*, *outline*, *text*, *pointName*, and *symbolName*. Including the layer name *allText* in the list suppresses the plotting of text; *bbox* suppresses the bounding box around symbols. *outline* suppresses the thin outline that borders each layer. The keywords *text*, *pointName*, and *symbolName* suppress the plotting of certain text created by local extension commands. *text* eliminates text created by user extension 2. *pointName* eliminates text created by user extension 94. *symbolName* eliminates text created by user extension 9. *allText*, *pointName*, and

*symbolName* may be abbreviated by *at*, *pn*, and *sn* respectively.

- c *n* (*copies*) Makes *n* copies of the plot. Works only for the Varian and Versatec. Default is 1 copy.
- d *n* (*depth*) This option lets you limit the amount of detail plotted in a hierarchically designed chip. It will only instantiate the plot down *n* levels of calls. Sometimes too much detail can hide important features in a circuit.
- g *n* (*grid*) Draw a grid over the plot with spacing every *n* CIF units.
- h (*half*) Plot at half normal resolution. (*Not yet implemented.*)
- e (*extensions*) Accept only standard CIF. User extensions produce warnings.
- I (*non-Interactive*) Do not ask for confirmation. Always plot.
- L (*List*) Produce a listing of the CIF file on standard output as it is parsed. Not recommended unless debugging hand-coded CIF since CIF code can be rather long.
- a *n* (*approximate*) Approximate a roundflash with an *n*-sided polygon. By default *n* equals 8. (I.e. roundflashes are approximated by octagons.) If *n* equals 0 then output circles for roundflashes. (It is best not to use full circles since they significantly slow down plotting.) (*Full circles not yet implemented.*)
- b "*text*" (*banner*) Print the text at the top of the plot.
- C (*Comments*) Treat comments as though they were spaces. Sometimes CIF files created at other universities will have several errors due to syntactically incorrect comments. (I.e. the comments may appear in the middle of a CIF command or the comment does not end with a semi-colon.) Of course, CIF files should not have any errors and these comment related errors must be fixed before transmitting the file for fabrication. But many times fixing these errors seems to be more trouble than it is worth, especially if you just want to get a plot. This option is useful in getting rid of many of these comment related syntax errors.
- r (*rotate*) Rotate the plot 90 degrees.
- N (*Printronic*) Send output to the Printronix.
- V (*Varian*) Send output to the Varian. (This is the default option.)
- W (*Wide*) Send output directly to the Versatec.
- S (*Speed*) Store the output in a temporary file then dump the output quickly onto the Versatec. Makes nice crisp plots; also takes up a lot of disk space.
- T (*Terminal*) Send output to the terminal. (*Not yet fully implemented.*)
- Gh (*Graphics terminal*) Send output to terminal using it's graphics capabilities. -Gh indicates that the terminal is an HP2648. -Ga indicates that the terminal is an AED 512.
- X *basename* (*eXtractor*) From the CIF file create a circuit description suitable for switch level simulation. It creates two files: *basename.slm* which contains the circuit description, and *basename.nod* which contains the node numbers and their location used in the circuit description.

When this option is invoked no plot is made. Therefore it is advisable not to use any of the other options that deal only with plotting. However, the -w, -l, and -a options are still appropriate. To get a plot of the circuit with the node numbers call *cifplot* again, without the -X option, and include *basename.nodes* in the list of CIF files to be

plotted. (This file must appear in the list of files before the file with the CIF End command.)

**-c *n*** (*copies*) This option specifies the number of copies of the plot you would like. This allows you to get many copies of a plot with no extra computation.

**-P *patternfile***

(*Pattern*) This option lets you specify your own layers and stipple patterns. *Patternfile* may contain an arbitrary number of layer descriptors. A layer descriptor is the layer name in double quotes, followed by 8 integers. Each integer specifies 32 bits where ones are black and zeroes are white. Thus the 8 integers specify a 32 by 8 bit stipple pattern. The integers may be in decimal, octal, or hex. Hex numbers start with '0x'; octal numbers start with '0'. The CIF syntax requires that layer names be made up of only uppercase letters and digits, and not longer than four characters. The following is example of a layer description for poly-silicon:

```
"NP" 0x08080808 0x04040404 0x02020202 0x01010101
      0x80808080 0x40404040 0x20202020 0x10101010
```

**-F *fontfilename***

(*Font*) This option indicates which font you want for your text. The *fontfilename* must be in the directory */usr/lib/vfons*. The default font is Roman 6 point. Obviously, this option is only useful if you have text on your plot.

**-O *filename***

(*Output*) After parsing the CIF files, store an equivalent but easy to parse CIF description in the specified file. This option removes the include and array commands (see next section) and replaces them with equivalent standard CIF statements. The resulting file is suitable for transmission to other facilities for fabrication.

In the definition of CIF provisions were made for local extensions. All extension commands begin with a number. Part of the purpose of these extensions is to test what features would be suitable to include as part of the standard language. But it is important to realize that these extensions are not standard CIF and that many programs interpreting CIF do not recognize them. If you use these extensions it is advisable to create another CIF file using the -O options described above before submitting your circuit for fabrication. The following is a list of extensions recognized by *cifplot*.

**0I *filename*;**

(*Include*) Read from the specified file as though it appeared in place of this command. Include files can be nested up to 6 deep.

**0A *s m n dx dy*;**

(*Array*) Repeat symbol *s* *m* times with *dx* spacing in the x-direction and *n* times with *dy* spacing in the y-direction. *s*, *m*, and *n* are unsigned integers. *dx* and *dy* are signed integers in CIF units.

**1 *message*;**

(*Print*) Print out the message on standard output when it is read.

**2 "*text*" transform ;**

**2C "*text*" transform ;**

(*Text on Plot*) *Text* is placed on the plot at the position specified by the transformation. The allowed transformations are the same as the those allowed for the Call command. The transformation affects only the point at which the beginning of the text is to appear. The text is always plotted horizontally, thus the mirror and rotate transformations are not really of much use. Normally text is placed above and to the right of the reference point. The 2C command centers the text about the reference point.

9 *name*;

(Name symbol) *name* is associated with the current symbol.

94 *name x y*;

94 *name x y layer*;

(Name point) *name* is associated with the point (*x*, *y*). Any mask geometry crossing this point is also associated with *name*. If *layer* is present then just geometry crossing the point on that layer is associated with *name*. For plotting this command is similar to text on plot. When doing circuit extraction this command is used to give an explicit name to a node. *Name* must not have any spaces in it, and it should not be a number.

#### FILES

~cad/.cadrc  
~/.cadrc  
~cad/bin/vdump  
/usr/lib/vfont/R.6  
/usr/tmp/#cif\*

#### SEE ALSO

cadrc(cad5)

A Guide to LSI Implementation, Hon and Sequin, Second Edition (Xerox PARC, 1980) for a description of CIF.

#### AUTHOR

Dan Fitzpatrick (UCB)

#### MODIFICATIONS

(UW/NW VLSI Consortium, University of Washington)

#### BUGS

The **-r** is somewhat kludgy and does not work well with the other options. Space before semi-colons in local extensions can cause syntax errors.

The **-O** option produces simple cif with no scale factors in the DS commands. Because of this you must supply a scale factor to some programs, such as the **-l** option to *cif2ca*.

The **-X** option does not work for non-manhattan circuits.

**NAME**

**lyra** - Performs hierarchical layout rule check on caesar design.

**SYNOPSIS**

```
lyra [-vz] [-o output] [-p path] [-r ruleset] [-t technology] rootCaesarFile,
or
lyra -e [-t technology] [-r ruleset]
```

**DESCRIPTION**

*Lyra* has two modes of operation: it can be invoked directly to perform a batch hierarchical check of a caesar design, or from the *Caesar* (or *Kic*) layout editor to interactively check a portion of the design currently being edited.

In batch mode, a hierarchical check of the caesar design rooted at *rootCaesarFile* is done. A log, including a summary of errors is written to stdout, and a *lyra file* "name.ly" is created for every cell "name.ca" in which design rule violations are detected. The lyra files flag each design rule violation with a bright splotch of paint on the error layer, and a caesar label identifying the type of violation. The lyra file for a cell "name.ca" contains the original caesar file as a subcell, thus the caesar subedit command can be used to conveniently fix design rule violations reported by *Lyra*. Obsolete lyra files are removed by *Lyra* when a cell checks on the current.run.

*Lyra's* violation messages have the form:

!< LayersOrConstructs > \_< Type > .

Note that all violation messages begin with an exclamation mark ("!"). *LayersOrConstructs* gives the single character abbreviations for the layers involved in the violation. Circuit constructs such as transistors and buried contacts may also be indicated by short abbreviations (e.g. tr for transistor; Be for buried contact). *Type* is given by one or two characters indicating the type of error as follows:

s = minimum spacing violation,  
w = minimum width violation,  
pe = parallel edge spacing violation,  
x = insufficient extension or enclosure,  
p = polarity, e.g. Dif. doping doesn't match well in CMOS,  
f = malformed circuit construct.

For example, a spacing violation between Polysilicon and Diffusion would look like this:

IP/D\_s.

Note that *Parallel Edge* checks are less restrictive than the corresponding *Width* and *Spacing* checks would be, since they ignore diagonal interactions.

The following *rulesets* are currently supported at Berkeley:

**nmosBERK**

Berkeley nMOS rules. Modified Mead & Conway rules. Buried contacts are supported; Butting Contacts are disallowed. The Lyon Implant rules are used.

**cmos-pwJPL**

CMOS rules (p well). An extension of the Mead and Conway nMOS rules to CMOS, worked out by Carlo Sequin in conjunction with JPL.

**nmosMC**

Mead & Conway nMOS rules as described in "Introduction to VLSI Systems" by Mead and Conway. Butting Contacts are allowed; buried contacts are not allowed.

**cmos-pw3**

MOSIS 3 micron bulk cmos process.

**isecmos**

GTE 5 micron isecmos process.

If the **-r** option is not given, *Lyra* chooses a *ruleset* based on the *technology* specified in the *rootCaesarFile*. The correspondence between *caesar technologies* and default *rulesets* is specified in *~cad/lib/lyra/DEFAULTS*. If *Lyra* does not recognize the *technology* of the *rootCaesarFile*, it uses the default *ruleset* for *nmos*.

In *editor mode* standard input and standard output are used to communicate with the layout editor, no log is written to *stdout*!, and violations are flagged directly in the edit cell. The *caesar technology* or *ruleset*, if different from *nmos*, must be specified explicitly on the command line, since *Lyra* does not have direct access to the *caesar* database. Note that interactive checks are nonhierarchical and slow, thus it is a good idea to use this mode only to check small pieces of a design; complete designs are best checked in batch mode.

The options described below may be specified in a *.cadrc* file or as command line options. *Lyra* reads options from *~cad/.cadrc*, */.cadrc* and the command line, in that order. If an option is specified in more than one place, the later setting takes precedence. Capitalizing an option on the command line, or giving the keyword *unset*<option> in *.cadrc* causes the option to be reset to its default value (e.g. "lyra -R", resets any previous *ruleset* specification, forcing the default to be used).

- e** (edit mode) Used by *Caesar* and *Kic*. In this mode *Lyra* reads rectangles etc. from standard input and reports violations on standard output.
- o <outputDir>**  
(output directory) Gives directory for *lyra* (-ly) files. Defaults to current directory.
- p <path>**  
(search path for caesar files) Path gives a colon (":") separated sequence of directories to be searched in order for *caesar* files. The default search path is just the current directory. As in *caesar* *~cad/lib/caesar* is searched as a last resort.
- r <ruleset>**  
(design rule set) Gives *ruleset* to use. *Rulesets* are stored in *~cad/lib/lyra*. A user can supply his own *ruleset* by giving the full pathname on the **-r** option (see *rulec*). If the **-r** option is not specified, *Lyra* determines which *ruleset* to use from the *technology* specified in the *rootCaesarFile* for the design.
- t <technology>**  
(caesar technology) Used to specify *caesar technology* in *editor mode*, or to override the *technology* given in the *rootCaesarFile*. *Lyra* uses the *caesar technology* to choose a default *ruleset*.
- v** (verbose mode) Causes more detailed log information to be written to *stdout*. This option is primarily for debugging.
- z** (restart) If *Lyra* dies abnormally, it leaves a *RESTART* file in the output directory which gives the cells which were completely checked. *Lyra* can then be restarted with the **-z** option, to resume checking with the first (sub)cell not already checked. Note that the restart option should only be used if the *caesar* database for the project has not been changed since the time the original *Lyra* run was started.



**FILES**

- \*cad/bin/lyra -- executable lyra.
- \*cad/lib/lyra -- rulesets (in symbolic and executable form).
- \*cad/lib/lyra/DEFAULTS -- gives default rulesets for caesar technologies.

**SEE ALSO**

Rulec (CAD)  
Caesar (CAD)  
KIC (CAD)  
Cif2ca (CAD)  
Cifplot (CAD)

**AUTHOR**

Michael Arnold.

**NAME**

**mextra** - Manhattan circuit extractor for VLSI simulation

**SYNOPSIS**

**mextra** [-g] [-u *scale*] [-o] *basename*

**DESCRIPTION**

*Mextra* will read the file *basename.cif* and create a circuit description. From this circuit description various electrical checks can be done on your circuit. The circuit description is directly compatible with *esim*, *powest*, and *erc*. There are translation programs to convert *mextra* output to acceptable *spice* input (see *sim2spice*, *pspice* and *spcpp*).

*Mextra* creates four new files, *basename.log*, *basename.al*, *basename.slm* and *basename.nodes*. After *mextra* finishes it is a good idea to read the *.log* file. This contains general information about the extraction. It has a count of the number of transistors and the number of nodes, and it contains messages about possible errors. The *.al* file is a list of aliases which can be used by *esim*. The *.nodes* file is a list of node names and their CIF locations listed in CIF format. It can be read by *cifplot* to make a plot showing the circuit with the named nodes superimposed. The form of this *cifplot* command is:

*cifplot* *basename.nodes* *basename.cif*

The *.slm* file is the circuit description for use with simulation programs and electrical rule checkers.

**Names**

*Mextra* uses the CIF label construct to implement node names and attributes. The form of the CIF label command is as follows:

**94** *name* *x* *y* [*layer*];

This command attaches the label to the mask geometry on the specified layer crossing the point (*x*, *y*). If no layer is present then any geometry crossing the point is given the label.

*Mextra* interprets these labels as node names. These names are used to describe the extracted circuit. When no name is given to a node, a number is assigned to the node. A label may contain any ASCII character except space, tab, newline, double quote, comma, semi-colon, and parenthesis. To avoid conflict with extractor generated names, names should not be numbers or end in '#*n*' where *n* is a number.

A problem arises when two nodes are given the same name although they are not connected electrically. Sometimes we want these nodes to have the same names, other times we don't. This frequently happens when a name is specified in a cell which is repeated many times. For instance, if we define a shift register cell with the input marked 'SR.in' then when we create an 8 bit shift register we could have 8 nodes names 'SR.in'. If this happens it would appear as though all 8 of the shift register cells were shorted together. To resolve this the extractor recognizes three different types of names: *local*, *global*, and *unspecified*. Any time a local name appears on more than one node it is appended with a unique suffix of the form '#*n*' where *n* is a number. The numbers are assigned in scanline order and starting at 0. In the shift register example, the names would be 'SR.in#0' through 'SR.in#7'. Global names do not have suffixes appended to them. Thus unconnected nodes with global names will appear connected after extraction. (The -g causes the extractor to append unique suffixes to unconnected nodes with the same global name.) Names are made local by ending them with a sharp sign, '#'. Names are global if they end with an exclamation mark, '!'. These terminating characters are not considered part of the name, however. Names which do not end with these characters are considered unspecified. Unspecified names are treated similar to locals. Multiple occurrences are appended with unique suffixes. By convention, unspecified names signify the designer's intention that this name is a local name, but is connected to only one node. It

is illegal to have a name that is declared two different types. The extractor will complain if this is so and make the name local.

It makes no difference to the extractor if the same name is attached to the same node several times. However, if more than one name is given to a node then the extractor must choose which name it will use. Whenever two names are given to the same node the extractor will assign the name with the highest type priority, global being the highest, unspecified next, local lowest. If the names are the same type then the extractor takes the shortest name. At the end of the .log file the extractor lists nodes with more than one name attached. These lines start with an equal sign and are readable by *esim* so that it will understand these aliases.

### Attributes

In addition to naming nodes *mextra* allows you to attach attributes to nodes. There are two types of attributes, *node attributes*, and *transistor attributes*. A node attribute is attached to a node using the CIF 94 construct, in the same way that a node name is attached. The node attribute must end in an at-sign, '@'. More than one attribute may be attached to a node. *Mextra* does not interpret these attributes other than to eliminate duplicates. For each attribute attached to a node there appears a line in the .slm file in the following form:

#### A node attribute

*Node* is the node name, and *attribute* is the attribute attached to that node with the at-sign removed.

Transistor attributes can be attached to the gate, source, or drain of a transistor. Transistor attributes must end in a dollar sign, '\$'. To attach an attribute to a transistor gate the label must be placed inside the transistor gate region. To attach an attribute to a source or drain of a transistor the label must be placed on the source or drain edge of a transistor. Transistor attributes are recorded in the transistor record in the .slm file.

### Transistors

For each transistor found by the extractor a line is added to the .slm file. The form of the line is:

```
type gate source drain length width x y
g=attributes s=attributes d=attributes
```

*Type* can be one of three characters, 'e' for enhancement, 'd' for depletion, or 'u' for unusual implant. ( Unusual implant refers to transistors which are only partially in an implanted area. It will be necessary to write a filter to replace these transistors with the appropriate model in terms of enhancement and depletion transistors.) *Gate*, *source*, and *drain* are the gate, source, and drain nodes of the transistors. *Length* and *width* are the channel length and width in CIF units. *X* and *y* are the x and y coordinates of the bottom left corner of the transistor. *Attributes* is a comma separated list of attributes. If no attribute is present for the gate, source, or drain, the g=, s=, or d= fields may be omitted.

The extractor guesses the length and width of a transistor by knowing the area, perimeter, and length of diffusion terminals. For rectangular transistors and butting transistors the reported length and width is accurate. For transistors with corners or for unusually shaped transistors the length and width is not as accurate.

It is possible to design a transistor with three or more diffusion terminals. The extractor considers these as *funny transistors*. They are entered in the .slm file in the form:

```
{type gate node1 node2 ... nodeN xloc
```

The 'T' is followed by the type : 'e', 'd' or 'u'. *Node1 ... nodeN* are the diffusion terminal nodes. As with any circuit with 'u' transistors, any circuit with 'T' transistors must be run through a filter replacing each of the funny transistors with the appropriate model in terms of enhancement and depletion transistors.

### Capacitance

The .slm file also has information about capacitance in the circuit. The lines containing capacitance information are of the form:

*C node1 node2 cap-value*

*cap-value* is the capacitance between a node and substrate in femto-farads. Capacitance values below a certain threshold are not reported. The default threshold is 50 femto-farads.

Transistor capacitances are not included since most of the tools that work on the .slm file calculate them from the width and length information.

The capacitance for each layer is calculated separately. The reported node capacitance is the total of the layer capacitances of the node. The layer capacitance is calculated by taking the area of a node on that layer and multiplying it by a constant. This is added to the product of the perimeter and a constant. The default constants are given below. Area constants are in femto-farads per square micron. Perimeter constants are femto-farads per micron.

Layer	Area	Perimeter
metal	0.03	0.0
poly	0.05	0.0
diff	0.10	0.1
poly/diff	0.40	0.0

Poly/diffusion capacitance is calculated similar to layer capacitance. The area is multiplied by constant and this is added to the perimeter multiplied by a constant. Poly/diffusion capacitance is not threshold, however.

The -e option suppresses the calculation of capacitance, and instead, gives for each node in the circuit the area and perimeter of that node on the diffusion, poly, and metal layers. The lines containing this information look like this:

*N node diffArea diffPerim polyArea polyPerim metalArea metalPerim*

*Node* is the node name. *x y* is the position of a point on the node. Currently this is always '0 0'. *DiffArea* through *metalPerim* are the area and perimeter of the diffusion, poly, and metal layers in user defined units. (In addition the -e option causes transistors with only one terminal to be recorded in the .slm file as a transistor with source connected to drain.)

*If the network is being extracted from the .cif file we suggest the node capacitance not be computed by mextra. Rather the -e option should be used. This puts the burden of computing node capacitance on the programs presim and sim2spice2. We feel this is advantageous because presim and sim2spice2 are filter programs linked directly to the type of simulation that is to be done. This will hopefully reduce some of the confusion associated with calibration.*

### Changing Default Values

As part of its start up procedure *mextra* reads two files: /usr/visibin/.cadrc and then a search for the first .cadrc from the current directory (.) to the the user's home directory is made. *Mextra* reads these files to set up constants to be changed without recompiling. The keywords for *mextra* are contained within the *mextra* environment of the .cadrc file. Declaration of environments in the .cadrc file are described in .cadrc(5).

By default, *mextra* reports locations in CIF coordinates. A more convenient form of units may be specified either in the .cadrc file or on the command line. The form of the line in the .cadrc file is:

**units** *scale*

where *scale* is in centi-microns. The user may type in the chosen value for the scale directly.

To set units on the command line use the **-u** option.

**mextra** **-u** *scale* *basename*

The parameters used to compute node capacitance may be changed by including the following commands in your **.cadrc** file.

**areatocap** *layer value*

**perimtocap** *layer value*

*value* is atto-farads per square micron for area, and atto-farads per micron for perimeter. *layer* may be "poly", "diff", "metal", or "poly/diff".

To set the capacitor values to those given in Mead and Conway the following lines would appear in the **.cadrc** file:

```
areatocap poly 40
areatocap diff 100
areatocap metal 30
areatocap poly/diff 400
perimtocap poly 0
perimtocap diff 0
perimtocap diff 0
perimtocap metal 0
perimtocap poly/diff 0
```

The threshold for reporting capacitance may be set in the **.cadrc** file with the following line.

**capthreshold** *value*

A negative value sets the threshold to infinity.

**Mextra** knows of two technologies, nMOS and cMOS p-well. NMOS is assumed by default. To set the technology to cMOS p-well, include the following line in your **.cadrc** file:

**tech** **cmos-pw**

## FILES

```
~/cadrc
basename.cif
basename.al
basename.log
basename.nodes
basename.sim
```

## SEE ALSO

```
powest(1.vlsi), pspice(1.vlsi), spcpp(1.vlsi), sim2spice(1.vlsi), spice(1.vlsi), drc(1.vlsi), erc(1.vlsi)
caesar(cad1),
cadrc(cad5),
```

## AUTHOR

Dan Fitzpatrick (UCB)

## MODIFICATIONS

(UW/NW VLSI Consortium, University of Washington)

## BUGS

Accepts manhattan simple CIF only, use **elfplot -O** to convert complicated CIF. For unusually shaped transistors the UW/NW modified **mextra** should be used, otherwise values will be quite inaccurate. The modified **mextra** will either yield accurate values or a "reasonable" guess,

depending on the complexity of the unusual transistor. The modified *mextra* will tell you when the output values are only best estimates. The length/width ratio for unusually shaped transistors may be inaccurate. This is true for snake transistors. Attributes for funny transistors are not recorded. Node attributes are ignored unless the *-e* switch is present.

**NAME**

**powest** - estimate power consumption of MOS circuits

**SYNOPSIS**

**powest** [-p] [*parm=value*] ...

**DESCRIPTION**

*Powest* reads a MOS circuit description in the format required by *esim(cad1)* and writes estimates of its DC power requirements.

Two types of input lines are accepted:

```
d gate source drain length width
e gate source drain length width
```

(Other lines are discarded.) These lines specify the connectivity and size of depletion (d) and enhancement (e) MOSFET's. The three signal names *gate*, *source*, and *drain* may be composed of any non-white space characters, and may be of arbitrary length. Upper and lower case distinctions are ignored only for the signals *Vdd* and *GND*. *Length* and *width* specify the channel dimensions in (integral) CIF units (centimicrons).

*Powest* looks for three types of pullup transistors:

- 1 Depletion device with its gate connected to its source, and its drain connected to *Vdd*.
- 2 Depletion device with its gate not directly connected to its source, and its drain connected to *Vdd* (e.g. depletion load of the second stage of a superbuffer)
- 3 Enhancement device with its gate and its drain both connected to *Vdd*.

(The symmetric cases in which the sources and drains are interchanged are also recognized.)

For each of these pullup types, a count of the number of such devices is given, along with an average and maximum DC power estimate attributable to those devices. For both of these it is assumed that *Vds* equals *Vdd* (or *Vsd* equals *Vdd* in the symmetric case), and for the depletion devices, *Vgs* (*Vsg*) equals zero. In obtaining the maximum estimate it is further assumed that the devices are on all the time. For the average power estimates it is assumed that the second type of depletion pullups and the enhancement pullups are on half the time. For the first type of depletion pullups, a count, *n*, is made of the number of enhancement devices connected to a pullup's source and it is assumed that the pullup is on a fraction  $1 - 2 \sup -n$  of the time.

The user may override the default values for the supply voltage and certain process parameters used in the calculations using the *parm=value* option. The -p option causes a list of these parameters and their effective values to be printed. Values take the form of C floating point constants. The following table summarizes the accessible parameters, their default values, and their implicit units.

<i>parm</i>	<i>value</i>	<i>units</i>	<i>description</i>
gamma	0.4	V <sup>0.5</sup>	bulk threshold parameter
tox	9.0e-8	m	oxide thickness
u0	0.08	m <sup>2</sup> /V-s	electron surface mobility
vsb	2.0	V	source-to-substrate voltage
vtb	-3.5	V	depletion threshold voltage
vte	0.8	V	enhancement threshold voltage
vdd	5.0	V	supply voltage

**SEE ALSO**

esim(cad1)

**AUTHOR**

Bob Cmelik (UCB)

**BUGS**

Pullups which don't fall into one of the three simple categories recognized can be the biggest power dissipators.



**NAME**

**sim2spice** - convert from mextra format to Spice (circuit simulator) format

**SYNOPSIS**

**sim2spice** [-d *defs*] *basename.sim*

**DESCRIPTION**

*Sim2spice* reads the *basename.sim*, *basename.nodes* and *basename.al* files created by *mextra* and creates a Spice readable circuit description, *basename.spice*. Spice requires node numbers and *sim2spice* generates a translation table *basename.names* which shows the *mextra* node label corresponding to a given node number.

The user can specify his/her own translation table by using the -d option, where *defs* is a file of definitions. A definition can be used to set up equivalences between *.sim* node names and Spice node numbers. The form of this type of definition is:

```
set sim_name spice_number [tech]
```

The *tech* field is optional. In nMOS, a special node, 'BULK', is used to represent the substrate node. For cMOS, two special nodes, 'NMOS' and 'PMOS', represent the substrate nodes for the 'n' and 'p' transistors, respectively. For example, for nMOS the *.sim* node 'GND' corresponds to Spice node 0, 'Vdd' corresponds to Spice node 1, and 'BULK' corresponds to Spice node 2. The *defs* file for this set up would look like this:

```
set GND 0 nmos
set Vdd 2 nmos
set BULK 3 nmos
```

A definition also allows you to set a correspondence between *.sim* transistor types and Spice transistor types. The form of this definition is:

```
def sim_trans spice_trans [tech]
```

Again, the *tech* field is optional. For nMOS these definitions would look as follows:

```
def e ENMOS nmos
def d DN MOS nmos
```

Definitions may also be placed in the *.cadrc* file, but the definitions in the *defs* file overrides those in the *.cadrc* file.

*Sim2spice* also reads 'N' lines generated by *mextra* with the -o switch. In order to compute capacitances from this it must have a set of conversion factors between length/area and capacitance. These are specified in the *sim2spice* section of *.cadrc* file in exactly the same format as in the *mextra* section of the *.cadrc* file (see *mextra*).

The program has been extended so that a comment line beginning with "!=" is interpreted as an MIT *.sim* style node equivalence line.

To create a complete Spice input file it is necessary to append applicable Spice model descriptions as well as the user's Spice simulation commands to the *basename.spice* file.

It is recommended in most cases that the user run *pspice* rather than *sim2spice*. *Pspice* incorporates the features of *sim2spice* but will in addition allow the user to build all of the Spice input file in one step. *Pspice* also incorporates the features of *spc*.

**FILES**

```
basename.sim
basename.nodes
basename.al
basename.spice
basename.names
```

**SEE ALSO**

mextra(1.vlsi), spice(1.vlsi), pspice(1.vlsi), spcpp(1.vlsi)

**AUTHOR**

Dan Fitzpatrick (UCB)

**MODIFICATIONS**

Neil Solffer (UCB) -- CMOS fixes.

Rob Fowler (UW/NW VLSI Consortium, University of Washington) -- node equivalence handling and misc. bug fixes.

**BUGS**

The only pre-defined technologies are 'nmos' and 'cmos-pw'. Only one definition file is allowed.

Warning: for nMOS circuits the node names "ENMOS" and "DNMOS" are preempted by *sim2spice* as synonyms for "BULK".

The node equivalence handling is not completely general. New nodes can be added to equivalence classes, but classes cannot be merged. This is detected and an error message is produced.

**NAME**

**tpla** – technology independent PLA generator

**SYNOPSIS**

**tpla** [-acv] [-s *style*] [-o *output\_file*] *input\_file*

**DESCRIPTION**

**Tpla** is a PLA generator that generates PLAs in several different styles and technologies. The input format is compatible with **eqntott**, see **PLA(5)** for details. **Tpla** does not handle split and folded PLAs.

**Tpla** is a program written with the **Tpack** system.

**STYLES OF PLAs AVAILABLE**

The following styles of PLAs are currently supported:

**Bcls** Buried contacts, nMOS, cis version (inputs and outputs on same side of the PLA). Clocked inputs and outputs are supported. Berkeley design rules.

**Btrans** Buried contacts, nMOS, trans version (inputs and outputs on opposite sides of the PLA). Clocked inputs and outputs are supported. Berkeley design rules.

**Mcls** Mead & Conway design rules. Butting contacts, nMOS, cis version (inputs and outputs on same side of the PLA). Clocked inputs and outputs are supported.

**Mtrans** Mead & Conway design rules. Butting contacts, nMOS, trans version (inputs and outputs on opposite sides of the PLA). Clocked inputs and outputs are supported.

**Tcls** Just like **Bcls** except that it has protection frames and terminals added (a special mod for EECS at Berkeley).

**Ttrans** Just like **Btrans** except that it has protection frames and terminals added.

**isocmos**

Complies with GTE 5 micron, isocmos process ( $\lambda = 2.5$  microns). Inputs and outputs on same side of PLA. Fabricated and tested.

**CS3cls** Complies with MOSIS 3 micron bulk CMOS process ( $\lambda = 1.0$  microns). Berkeley design, simulated but not fabricated. Inputs and outputs on same side of the PLA.

**CS3tran**

Same as **CS3cls** except inputs and outputs on opposite sides of the PLA.

It is easy to create a template for a new style of PLA, and **tpla(CAD5)** has information on how to do it. If you develop a particularly nice template and would like to share it, send it to "mayo@berkeley" or "ucbvax:mayo".

**Tpla** handles CIF symbol naming directives and input & output labels as described in **pla(CAD5)**.

**OPTIONS**

**-I** Clock the inputs to the PLA, if this feature is supported for this style.

**-O** Clock the outputs to the PLA, if this feature is supported for this style.

**-G num** Insert an extra ground line every *num* rows in the AND plane and every *num* columns in the OR plane.

**-S num** Stretch power and ground lines by *num* lambda.

**-v** Be verbose, and show (in the Caesar output) how the PLA was constructed from its basic components.

- V Be verbose, and print out information about what tpla is doing. This option implies -v.
- a produce Caesar format (this is the default)
- c produce CIF format
- o The next argument is taken to be the base name of the output file. The default is the input file name with any extensions removed. If the input comes from the standard input and the -o option is not specified then the output will go to the standard output.
- s The next argument specifies the style of PLA to generate. (This causes tpla to use the file `~cad/lib/tpla/p-style.tp` as its template).
- l *num* Set lambda to *num* centimicrons. (200 is the default)
- t The next argument specifies the template to use, this normally defaults to the standard library. A .tp suffix is added if no suffix was specified. This option is useful for generating styles of PLAs that are not included in the standard library.

**input\_file**

The file containing the truth\_table. If this filename is omitted then the input is taken from the standard input (such as a pipe).

**other options**

This program inherits several more options from Tpack(CAD).

**FILES**

<code>~cad/bin/tpla</code>	-- executable
<code>~cad/src/tpla/*</code>	-- source
<code>~cad/lib/tpla/p*.tp</code>	-- standard templates for PLAs

**SEE ALSO**

`eqntott(CAD)`, `presto(CAD)`, `plasort(CAD)`, `pla(CAD5)`, `tpla(CAD5)`, `tpack(CAD)`, `mkpla(CAD)`

**AUTHOR**

Robert N. Mayo

**BUGS**

The defaults for the -G and -S options have no way of knowing what the grounding requirements are for the style of PLA actually being generated.

This program inherits any bugs that may exist in `tpack(CAD)`.

# PLAP User's Guide

UW/NW VLSI Consortium  
University of Washington  
Seattle, WA 98195

This manual corresponds to PLAP version 3.0.

## 1. Introduction

LAP is a package of artwork generation routines, originally written at Caltech by Bart Locanthi. It generates integrated circuit artwork data files in the Caltech *Intermediate Form (CIF)* in response to procedure calls executed within a program written by the VLSI designer. Boeig **plap** is a complete re-design of the original software. It is written in PASCAL rather than SIMULA, has many additions to improve the user interface, and removes former restrictions of maximum chip complexity.

**plap** allows either the entire design, a group of cells (symbols), or a single cell to be contained in a program. Thus, if only one symbol needs to be altered, only the program containing that symbol needs to be re-compiled and re-executed. The result is a minimization of the compilation and execution times required to design an integrated circuit, assuming a careful decomposition of the design.

Also, **plap** has the ability to read cells which have been created using **caesar**, the Berkeley interactive graphics layout package, and can create **caesar** format cells as well. These cells will have a *.ca* extension. Typically in the design process, the designer will lay out the low level cells using **caesar** and will place and interconnect them with **plap**. It must be pointed out that using this approach limits the designer to Manhattan geometries only, as **caesar** will not allow non-Manhattan geometry.

If the user is not creating **caesar** files using **plap**, the files created will be in the db format. There will be two files created for each symbol defined in the **plap** code. The first is the *.sym* file, which contains modified *CIF* code with symbol numbers replaced by symbol names. The second is the *.ast* file, which consists of bounding box coordinates, a list of the symbols called, and a list of user-definable attributes. These attributes are (name,value) pairs where the attribute name is associated with an arbitrary real number. They may be used at the designer's discretion and routines are provided for setting and retrieving attributes. Both symbol files are read-only protected to prevent any manual editing of their contents. To produce a "standard" *CIF* file from files in the db format, the **db2cif** conversion program must be run. This collects all the *.sym* files required to completely describe the given symbol. In creating *CIF* from a **caesar** format file, see the documentation on **caesar**.

**plap** uses macro calls to lay out certain symbols which have been made available so as to facilitate the layout process. These macros will position and orient certain specific symbols contained in the symbol library. Also, there other symbols in the library which can be used in designs and can be positioned

using the more general placement macros.

In order to facilitate the use of a large data base of symbol files, **plap** makes use of a symbol data base organized as cell libraries. There must be at least *one*, but not more than *three* libraries declared in a **plap** program. The current working directory should never be used as one of the libraries to avoid a clutter of **plap** programs and symbol files. The symbol files will be created in the last library declared in the **plap** program.

## 2. How to Use Boeing **plap** v.3

### 2.1. Program Shell

A small shell file is provided which sets up the proper global variable environment for linking with the **plap** modules. User-defined variables and procedures must be inserted in appropriate places *following* the **plap** system declarations. This shell file is located in `$UW_VLSI_TOOLS/lib/psh` under the name `tek.psh`, with `tek` being either `nmos`, `isocmos`, `cmospw`, or `i2l`. The user should be aware that `$UW_VLSI_TOOLS` is an environment variable specifying the path to the source directory for the vlsi design tools. The template file `$UW_VLSI_TOOLS/lib/psh/isocmos.psh` is shown below.

```
{ .....}
{ * basic shell for VLSI artwork design language * }
{ .....}

{ * program shell * }

#include '$UW_VLSI_TOOLS/include/plap3/isocmosext.h'

procedure user;          { insert user-defined const, type, var here }

begin {** main program **}

{ declare up to three libraries }
  initialize('project');
  library('$UW_VLSI_TOOLS/lib/celllib/isocmos/');
  library('lib/');

{ .....}
{ * insert VLSI design definitions here * }
{ .....}

{ .....}
{ * end of user VLSI design definitions * }
{ .....}

end. {** main program **}
```

The `#include` statements cause external text files to be inserted at the locations which contain the actual declarations. These files must be the same files used to compile the **plap** system modules. Users should verify that the pathnames are the ones actually used on their system.

The *initialize* and *library* calls must precede any other main program statements.

Note: On UNIX, Pascal keywords must be in lower case and the case of variables *is* significant. Therefore, users should be aware of the possibility of this type of error.

## 2.2. Compile, Link, and Run

The *plap* design file is a Pascal program which must be compiled, linked with external *plap* modules, and executed. A command file is provided which performs all of the above for the user. Simply type:

```
% plap [-options] basename
```

The file extension is not necessary in *basename*.

*Options* includes :

**-norun**

Prevents execution of the compiled *plap* program.

**-nocompile**

Prevents compilation of the *plap* program.

**-nmos**

**-isocmos**

**-cmospw**

**-i2l**

Design is being done in the respective technology, either *nmos*, *isocmos*, *cmospw* or *i2l* (default is *nmos*).

**-sym**

Files have db database format (default), with file extensions of *.sym* and *.ast*.

**-ca**

Files have caesar database format, with file extensions of *.ca*.

The *plap* program will execute, listing symbol names on the terminal as they are defined and displaying any error messages that might occur. All information displayed on the terminal is also echoed into the log file whose name was entered in the *initialize* statement. In addition, *plap*'s interpretation of any PLA code files will be echoed into the log file for confirmation by the user.

After processing the user's program, *plap* will give the prompt:

```
menu: w=write, s=show, q=quit, a=abort?
```

Typing *s*<return> will cause all the symbols in the symbol tree to be listed, along with the bounding box of each symbol. Typing *q*<return> will terminate the program if a write has already been performed. Typing *a*<return> will terminate the program without producing any *.sym* and *.ast* or *.ca* files. Typing *w*<return> will cause the *.sym* and *.ast* or *.ca* files to be written onto the disk in the user's account. They will be written into the last library declared by a *library* call in the *plap* design file.

## 2.3. Units and Parameter Types

All dimensions are in lambda (Mead & Conway). Conversion to microns is done when the CIF file is made using *db2cif* or *caesar*.

Parameters passed to *plap* commands must adhere to certain Pascal type-checking rules. For instance, all dimensions or coordinates must be of type *real*. Other parameters are of types such as *integer* or *string*, as described for each command in Sections 3 and 4 of this manual.

---

†UNIX is a Trademark of Bell Laboratories.

Parameters can be literal constants (numbers) as well as variables. Under UNIX, real numbers need a decimal point and at least one digit preceding and following the decimal point. (The UNIX Pascal compiler may create illegal binary code if real constants are not followed by a decimal point and one digit.)

A string can be either a literal enclosed in single quotes, or a variable. Strings can be any length, within the following constraints:

- String parameters specifying symbol names (in `define` and `draw` commands) are truncated to 9 characters. Other text string parameters may be up to 80 characters in length.
- Strings do not need to be padded with blanks to a fixed length, and trailing blanks are ignored.
- String parameters must be at least 2 characters, however the second character may be a blank, which would then be ignored.

### 3. Detailed Discussion of Commands

#### 3.1. Artwork Primitives

`layer('layerName');`

This command sets the mask layer for succeeding art work. `plap` understands the layer names to be those specified in the `technology.tec` file. The following is a list of those, and their corresponding CIF layers, for *nMOS*, *ISOCMOS*, *CMOSPW*, and *I<sup>2</sup>L* technologies:

*nMOS* Parameters:

<i>'layerName'</i>	CIF layer
polysilicon	NP
metal	NM
metal2	NM2
diffusion	ND
implant	NI
himplant	NH
contact	NC
contact2	NC2
glass	NG
buried	NB
extra	NX

*ISOCMOS* Parameters:

<i>'layerName'</i>	CIF layer
poly	CP
poly2	CP2
metal	CM
metal2	CM2
diff	CD
pplus	CPP
nplus	CNP
pwell	CPW
pads	XP
label	LL
glass	CG



**CMOSPW Parameters:**

cut	CC
'layerName'	CIF layer
polysilicon,poly	CP
metal	CM
metal2	CM2
diffusion,diff	CD
pwell	CW
pplus	CS
glass	CG
label	LL
pads	XP

**I<sup>2</sup>L Parameters** (note : No support will be offered for this technology in later releases) :

'layername'	CIF layer
metal	IM
metal2	IM2
emitter	IE
base	IB
sinker	IS
glass	IG
contact	IC
contact2	C2
buried	IBL
isolation	II
resistor	IR

**Examples:**

```
layer('polysilicon');
layer('metal');
layer('sinker');
```

```
box(x1,y1,x2,y2);
```

```
lbox('layerName',x1,y1,x2,y2);
```

**layerName** Layer name as described for the layer command.

**x1,x2,y1,y2** Coordinates in lambda units (real);

Both of these procedures draw a box with lower left corner at (x1,y1) and upper right corner at (x2,y2). Box edges are always aligned with respective x- and y-axes.

The lbox procedure causes a permanent change in the mask layer before the box is generated.

Although a warning message is produced if a box has zero width or length, it is still output to the CIF file.

**Examples:**

```
box(-2.9,7.0,9.5,26.0);
lbox('metal',13.0,0.5,19.0,9.5);
```

**wire**(width,xStart,yStart); **wirePath**  
**lwire**('layerName',width,xStart,yStart); **wirePath**

**layerName** Layer name as described for the layer command;  
**width** Width in lambda units (integer);  
**xStart,yStart** Coordinates in lambda units (real).  
**wirePath** An arbitrary sequence of wire path primitive commands (Section 3.2). A wire path is automatically terminated by any **plap** command that is not one of these primitives.

Both of these commands draw a wire of width=**width** starting at (xStart,yStart) and continuing as specified in the **wirePath**. The **wire** procedure draws the wire in the same layer as last drawing command. The **lwire** procedure changes the layer to **layerName** before drawing the wire.

Examples:

```
wire(2.0,-7.0,2.5);
  x(23.0);y(7.0);xy(35.0,17.5);x('metal');dx(-4.0);
lwire('metal',3.0,9.0,6.0);
  dxy(12.0,9.0);w(10.0);x(21.0);y(17.5);
```

**polygon**(xStart,yStart); **polyPath** {**polyclose**};

**xStart,yStart** Coordinates in lambda units (real);  
**polyPath** An arbitrary sequence of wire path statements, *excluding* the **x** and **w** commands (Section 3.2).  
**polyclose** An optional command to close the polygon (generate a final vertex equal to the first vertex). If a polygon is not closed or has less than three sides, an error message will be generated.

This procedure generates an arbitrary polygon. *It should be used only if absolutely necessary and when the user is willing to take the risk that downstream analysis programs may not accept polygons.* For instance, **fastdre**, **alldre** and the **maxtra** extractor ignore all polygons.

If a polygon statement is used anywhere in the program, a warning message will be given at the end of program execution.

Examples:

```
polygon(1.0,3.5);
  x(5.0);dy(2.5);x(-4.0);y(-2.0);x(1.0);y(3.5);
polygon(1.0,3.5);
  xy(5.0,6.0);x(-4.0);y(-3.5);x(1.0);polyclose;
```

### 3.2. Path Primitives

**xy**(xNext,yNext);  
**dxy**(xDelta,yDelta);  
**x**(xNext);  
**y**(yNext);  
**dx**(xDelta);  
**dy**(yDelta);  
**xNext,yNext** Coordinates in lambda units (real);

*xDelta,yDelta* Lengths in lambda units (real);

Procedures *x*, *y* and *xy* create a wire from the current position to the point with the specified new *x*- and/or *y*-coordinate.

Procedures *dx*, *dy* and *dxy* create a wire from the current position to the point with the specified change to the *x*- and/or *y*-coordinate.

*w(newWidth)*;

*newWidth* Width in lambda units (integer);

The *w* procedure changes the current wire width to be *newWidth*. Subsequent commands in the wire path will create wires of this width, until changed again. This command is only appropriate in a *wirePath*, not in a *polyPath*.

### 3.3. Symbol Definition and Call Statements

```
define('symbolName');
enddef;
```

*symbolName* A string constant or variable (see Section 2.3 for a description of acceptable strings).

This pair of statements is used to bracket a symbol definition. A symbol must be defined before it can be used. All artwork statements must be contained within symbol definitions, otherwise a warning message will be produced.

The symbol name must be unique and must not correspond to any of the symbol names in the technology library. Symbol definitions may not be nested. This restriction is due to *CIF* not allowing nested definitions.

Examples:

```
define('latch');
.
.
enddef;

define('REGISTER_16');
.
.
enddef;
```

```
draw('symbolName',x,y);
drawmx('symbolName',x,y);
drawmy('symbolName',x,y);
drawrot('symbolName',x,y,vx,vy);
```

*symbolName* Symbol name as described for the *define* command.

*x,y* Coordinates in lambda units (real);

*vx,vy* Coordinates (integer) of the vector endpoint starting at the origin;

These procedures cause a symbol with name *symbolName* to be drawn with its origin at (*x,y*). The symbol must be defined before it can be drawn.

*drawmx* first mirrors the symbol in the *x*-dimension (exchanges *-x* for *x*) and *drawmy* first mirrors the symbol in the *y*-dimension (exchanges *-y* for *y*). *drawrot* first rotates the symbol using the rotation vector(*vx,vy*). The symbol is rotated through the same angle that the rotation vector is rotated from the *x*-axis. For instance, the following rotation vectors correspond to the these

rotations:

( 1, 0)	no rotation
( 0, 1)	90 degrees
(-1, 0)	180 degrees
( 0,-1)	270 degrees
( 1, 1)	45 degrees

The rotation vector can be of any magnitude except zero. *The user should be warned that non-orthogonal rotations may cause difficulty in downstream analysis programs and thus should be avoided.*

Examples:

```
draw('latch',23.0,37.5);
drawmx(pla1'.planame,0.0,-23.0);
drawrot('datapaths',7.0,13.0,0,1);
```

### 3.4. Functions and Annotation Statements

lastX;

lastY;

These functions return the most recent *x* or *y* coordinate (real) used in a series of wire path statements. *box* or *draw* commands do not set these values.

Examples

```
wire(2.0,7.0,13.0);x(11.0);y(21.0);z('metal');
wire(2.0,lastX,lastY);x(-2.0);dy(-2.0);
```

plottext('textString',*x,y*,*centered*);

nodelabel('textString',*x,y*,*layerName*);

*textString*    A string constant or variable, up to 80 but at least 2 characters in length. Trailing blanks are ignored.

*x,y*            Coordinates in lambda units (real);

*centered*      Boolean flag;

*layerName*    Layer name as described for the *layer* command.

These statements serve to annotate the Versatec plots created by the Berkeley *clplot* program, and to name nodes for the *mextra* extractor.

*plottext* plots a string of text at (*x,y*) on the Versatec plot. If *centered* = "true", the text is centered on (*x,y*); otherwise (*x,y*) is the lower left corner of the text. The text is always horizontal. This is useful for labeling the plot with date and version of the design.

*nodelabel* plots a text string at (*x,y*) on the Versatec plot and attaches a node name to any feature intersecting (*x,y*) on layer *layerName*. For *clplot*, (*x,y*) is the lower left corner of the text and *layerName* is significant only in that if that layer is excluded from the plot, the string will also be excluded. Node labels are required for the *mextra* extractor, *erc* and *messim* simulator. (See the user's guides for those programs.)

Examples:

```
plottext('version8/23/81',100.0,239.0,false);
nodelabel('muxinput',10.0,22.5,'metal');
```

### 3.5. Boolean Switch Variables

```
checkswitch := boolean;
manhattan   := boolean;
halfcheck    := boolean;
```

These are global Boolean variable switches that the user may override.

**checkswitch** False by default, but if set to true, causes the program to check for minimum wire widths in all wires.

**manhattan** True by default and causes the program to check that all geometry in the user's design contains only 90 degree angles.

**halfcheck** True by default and causes the program to check that all figures in the user's design fall only on 1/2 lambda boundaries.

Any of the above switches may be set by assignment statements placed after the `initialise` statement and before any user VLSI design functions.

Examples:

```
checkswitch := true;
manhattan   := false;
halfcheck    := false;
```

### 3.6. Macros

```
downroute(nLines,xStartArray,yStart,xEndArray,yEnd,width,spacing);
```

***nLines*** Number of wires to route (integer), where  $nLines \leq 50$ ;

***yStart*,*yEnd*** y-coordinate in lambda units (real);

***xStartArray*,*xEndArray*** Array of *nLines* x-coordinates in lambda units (real). This is a predefined type called `routarray`;

***width*** Width in lambda units (integer);

***spacing*** Minimum wire spacing in lambda units (integer).

This procedure performs a simple river-routing in the current mask layer, with right-angled corners. It only works in the vertical direction, preferably top-down (bottom-up may still have some bugs). It connects an array of wires, all starting at *y=yStart*, with initial x-coordinates in *xStartArray*, to points at *y=yEnd* with final x-coordinates in *xEndArray*. All x-coordinates in the two arrays must be in ascending order, and *no crossings are allowed*. The wires will be drawn with width and spacing as specified by the last two parameters.

Example:

In the declarations section:

```
var startxs,endxs: routarray;
```

In the execution section:

```
layer('polysilicon');
for i := 1 to 23 do
begin
  startxs[i] := 8*i;
  endxs[i] := 102 + 9*i;
end;
downroute(23,startxs,0.0,endxs,-120.0,2,2);
```

**alpha('textString',x,y,scale);**

**textString** A string constant or variable up to 80 characters long;

**x,y** Coordinates in lambda units (real);

**scale** Text scaling factor (real).

This macro call uses wires to generate letters in the current mask layer. (x,y) is the lower left corner of the string, and the letters are created with

height =  $8 \times \text{scale}$  (lambda),

width =  $5 \times \text{scale}$  (lambda),

spacing =  $2 \times \text{scale}$  (lambda),

thickness =  $\text{scale}$  (lambda).

The following characters are supported:

A..Z, 0..9,

-. , ? / { } + =

Lower case is converted to upper case, but any other character will result in a blank space.

The alpha command is very expensive in computer time and CIF file space. *It should not be used for extensive plot annotation.* For plot annotation, use **plottext** or **modelabel**.

Examples:

**alpha('8X8MultiplierVersion2/17/81',103.0,79.5,2.0);**

**alpha('date-out',5.0,27.0,3.0);**

### 3.7. Symbol Access Statements

**library('libraryName');**

**libraryName** A full Unix path name.

The library procedure is used to establish a search path for **plap** to follow when looking for a particular symbol. In searching for a symbol, **plap** will search the libraries in the reverse order in which they were declared and it stops as soon as it finds the first occurrence of the symbol. If the symbol is not found, **plap** will search its library the standard cell library, located in **SUW\_VLSI\_TOOLS/src/celllib/tek/**, for the symbol. *tek* refers to the particular technology being used for the design, which is a directory in which the symbols are located. The user *must* declare his own working library and he may *optionally* declare group project and system libraries. The libraries must be declared before any symbols are defined. A separate library command must be used for each of up to *three* library declarations.

Example:

```
begin
  initialize(...);
  library('/u1/larry/lib');
  .
  .
end.
```

**withsym('symbolName');**

**symbolName** Symbol name as described for the **define** command.

This procedure is used in conjunction with the **fa** function and **setatt** procedure to specify the symbol currently being accessed.

Example:

```
var x : real;

begin
  initialize(...);
  withsym('cell');
  x := fa('vddx');
  setatt('vddy',0.0);
  .
end.
```

In this example, the symbol being accessed is "cell". The call to fa will set variable x to be equal to the "vddx" value from "cell" and the call to setatt will set the "vddy" attribute for "cell1" to be zero.

**setatt('attributeString',value);**

**attributeString**     An attribute name as described for the fa function.

**value**                The value (real) to be assigned to the attribute name.

This procedure sets the attribute name for a symbol to have the corresponding value.

Example:

```
begin
  initialize(...);
  .
  withsym('cell2');
  setatt('cell1.gndx',15.5);
  setatt('gndx',10.5);
  .
end.
```

The first call to setatt will set the attribute "gndx" of the symbol "cell1" to have the value 15.5 and the second call will set the attribute "gndx" of the symbol "cell2" to be 10.5.

**fa('attributeString');**

**attributeString** may be one of the following:

1. A string containing only the attribute name, in which case the attribute associated with the symbol name used in the last call to the withsym procedure.
2. A string of the form *symbolName.attributeName*, in which case the attribute is assumed to be associated with the symbol name specified in the string.

The fetch attribute function returns the value of an attribute of a symbol as a real number. The attribute must have been set beforehand by the setatt procedure.

Labels created using **casar** can be accessed by appending either **'\_x'** or **'\_y'** to the attribute name.

Example:

```

var x1,x2 : real;

begin
  initialize(...);
  .
  .
  withsym('cell1');
  x1 := fa('gndx');
  x2 := fa('cell2.gndx');
  .
end.

```

In the example, x1 will be set to the value of "gndx" from "cell1" and x2 will be set to the value of "gndx" from "cell2".

#### 4. ISOCMOS Macros

*Note : This is the GTE ISOCMOS process which assumes a minimum polysilicon wire width of 2 lambda.*

##### 4.1. Basic Macros

```

mp(x,y);
md(x,y);
be(x,y);
bn(x,y);
bs(x,y);
bw(x,y);
op(x,y);
on(x,y);

```

*x,y* Coordinates in lambda units (real).

These macro calls generate symbol calls for various ISOCMOS contacts. *mp* and *md* are metal-polysilicon and metal-diffusion contacts, respectively, centered at (*x,y*). The next four calls generate butting contact symbol calls in various orientations. The butting contact origin is one lambda from the diffusion end, and is placed at (*x,y*). The last two calls deal with the placement of ohmic contacts, again centered at (*x,y*). *op* places an ohmic contact on p type material, and *on* places an ohmic contact on n type material. *be* places the polysilicon end eastward, *bn* northward, etc.

Examples:

```

mp(5.0,22.5);
be(12.0,19.5);
on(34.5,26.5);

```

#### 5. CMOSPW Macros

*Note : This is the MOSIS bulk cmos process with 3 micron rules, where the minimum polysilicon wire width is 3 lambda.*



### 5.1. Basic Macros

**gb(x,y);**

**rb(x,y);**

*x,y*    Coordinates in lambda units (real).

These macro calls generate symbol calls for various CMOSFW contacts. **gb** and **rb** are green-blue (diffusion-metal) and red-blue (polysilicon-metal) contacts, respectively, centered at (x,y).

Examples:

**gb(5.0,22.5);**

**rb(12.0,19.5);**

## 6. nMOS Macros

*note : All nMOS macros assume a polysilicon wire width of 2 lambda, and use Mead-Conway design rules.*

### 6.1. Basic Macros

**z('newLayerName');**

*newLayerName*    Layer name as described for the **layer** command;

The **z** procedure changes the current layer to be *newLayerName*. A contact will be placed at the current point in order to make the change from the current layer to the new layer specified. Subsequent commands in the wire path will create wires on this layer, until changed again. This command is only appropriate in a *wirePath*, not in a *polyPath*.

**rb(x,y);**

**gb(x,y);**

*x,y*    Coordinates in lambda units (real).

These macro calls generate symbol calls for various nMOS contacts. **rb** and **gb** are red-blue (polysilicon-metal) and green-blue (diffusion-metal) contacts, respectively, centered at (x,y).

Examples:

**rb(26.0,39.5);**

**bn(-7.5,2.0);**

## 7. Technology Cell Libraries

Certain pre-generated symbols are available for use in designing with nMOS, CMOSFW, and ISOC-MOS layouts. These can be used with the various symbol manipulation commands available in **plap**. The user should avoid using the same names of these symbols when designing his own symbols. A more complete description is contained in the **Standard Cell Library Guide**.

## 8. Command Summary

### 8.1. Artwork Primitives

```
layer('layerName');
box(x1,y1,x2,y2);
lbox('layerName',x1,y1,x2,y2);
wire(width,xStart,yStart); wirePath
lwire('layerName',width,xStart,yStart); wirePath
polygon(xStart,yStart); polyPath {polyClose;}
```

### 8.2. Path Primitives

```
xy(xNext,yNext);
dxy(xDelta,yDelta);
x(xNext);
y(yNext);
dx(xDelta);
dy(yDelta);
w(newWidth);
```

### 8.3. Symbol Definition and Call Statements

```
define('symbolName');
endef;
draw('symbolName',x,y);
drawmx('symbolName',x,y);
drawmy('symbolName',x,y);
drawrot('symbolName',x,y,vx,vy);
```

### 8.4. Functions and Annotation Statements

```
lastX;
lastY;
plottext('textString',x,y,centered);
nodelabel('textString',x,y,'layerName');
```

### 8.5. Boolean Switch Variables

```
checkswitch
manhattan
halfcheck
```

### 8.6. Macros

```
downroute(nLines,xStartArray,yStart,xEndArray,yEnd,
width,spacing);
alpha('textString',x,y,scale);
```

### 8.7. Symbol Access Statements

```
library(libraryName);
withsym(symbolName);
setatt(attributeString,value);
fa(attributeString);
```

**8.8. nMOS Macros**

```
z('newLayerName');  
rb(x,y);  
gb(x,y);
```

**8.9. ISOCMOS Macros**

```
mp(x,y);  
md(x,y);  
be(x,y);  
bn(x,y);  
bs(x,y);  
bw(x,y);  
op(x,y);  
on(x,y);
```

**8.10. CMOSPW Macros**

```
gb(x,y);  
rb(x,y);
```

**9. ISOCMOS Example**

The following is a sample file of an ISOCMOS layout. It is a bidirectional inverting buffer using cells from the ISOCMOS Standard Cell Library, as described in the document **Standard Cell Library Guide**. It uses two cells from the Standard Cell Library, an inverter and two clocked inverters for a total of three devices. In the example below, user text is "in boldface" for highlighting purposes.

```

#include '$UW_VLSI_TOOLS/include/plap3/isocmosext.h'

(* insert user-defined const, type, var here *)
var
  xw0, yw0, xw1, yw1, xw2, yw2, ox, oy,

  vddx0, vddy0, gndx0, gndy0,
  lnx0, lnty0, lnby0,
  outx0, outty0, outby0,
  clkx0, clkby0, clkty0,
  clkbrx0, clkbrby0, clkbrty0,

  vddx1, vddy1, gndx1, gndy1,
  lnx1, lnty1, lnby1,
  outx1, outty1, outby1,

  vddx2, vddy2, gndx2, gndy2,
  lnx2, lnty2, lnby2,
  outx2, outty2, outby2,
  clkx2, clkby2, clkty2,
  clkbrx2, clkbrby2, clkbrty2,

  metalwidth, polywidth, sep01, sep12

: real ;

begin

  initialize('project');
  library('$UW_VLSI_TOOLS/src/celllib/isocmos/');
  library('lib/');

  sep01 := 2.0 ; sep12 := 2.0 ;
  metalwidth := 2.0 ; polywidth := 2.0 ;
  ox := 0.0 ; oy := 0.0 ;
  define('bidirect') ;
    draw('clkinv',ox,oy) ;

  (* Obtain coordinate information of the labels contained in
  the clocked inverter cell. *)
  withsym('clkinv') ;
  xw0 := fa('upperRight_x') ; yw0 := fa('upperRight_y') ;

  lnx0 := fa('lnb_x') ;
  lnby0 := fa('lnb_y') ; lnty0 := fa('lnt_y') ;

  clkbrx0 := fa('clockBarb_x') ;
  clkbrby0 := fa('clockBarb_y') ; clkbrty0 := fa('clockBart_y') ;

  clkx0 := fa('clockb_x') ;
  clkby0 := fa('clockb_y') ; clkty0 := fa('clockt_y') ;

  outx0 := fa('outb_x') ;
  outby0 := fa('outb_y') ; outty0 := fa('outt_y') ;

```

```
vddx0 := fa('vddRight_x') ; vddy0 := fa('vddRight_y') ;
gndx0 := fa('gndLeft_x') ; gndy0 := fa('gndLeft_y') ;
```

(\* Draw a basic inverter cell \*)

```
draw('inv', ex + xw0 + sep01, ey) ;
```

(\* Obtain coordinate information of the labels contained  
in the inverter cell. \*)

```
withsym('inv') ;
```

```
xw1 := fa('upperRight_x') ; yw1 := fa('upperRight_y') ;
```

```
inx1 := fa('inb_x') + xw0 + sep01 ;
```

```
inby1 := fa('inb_y') ; inty1 := fa('int_y') ;
```

```
outx1 := fa('outb_x') + xw0 + sep01 ;
```

```
outby1 := fa('outb_y') ; outty1 := fa('outt_y') ;
```

```
vddx1 := fa('vddRight_x') + xw0 + sep01 ; vddy1 := fa('vddRight_y') ;
```

```
gndx1 := fa('gndLeft_x') + xw0 + sep01 ; gndy1 := fa('gndLeft_y') ;
```

(\* Lay the second clocked inverter out. \*)

```
draw('clkinv', ex + xw0 + sep01 + xw1 + sep12, ey) ;
```

(\* Obtain coordinate information of the labels contained in  
the clocked inverter cell. \*)

```
xw2 := xw0 ; yw2 := yw1 ;
```

```
inx2 := inx0 + xw0 + xw1 + sep01 + sep12 ;
```

```
inby2 := inby0 ; inty2 := inty0 ;
```

```
clkbrx2 := clkbrx0 + xw0 + xw1 + sep01 + sep12 ;
```

```
clkbrby2 := clkbrby0 ; clkbrty2 := clkbrty0 ;
```

```
clkx2 := clkx0 + xw0 + xw1 + sep01 + sep12 ;
```

```
clkby2 := clkby0 ; clkty2 := clkty0 ;
```

```
outx2 := outx0 + xw0 + xw1 + sep01 + sep12 ;
```

```
outby2 := outby0 ; outty2 := outty0 ;
```

```
vddx2 := vddx0 + xw0 + xw1 + sep01 + sep12 ; vddy2 := vddy0 ;
```

```
gndx2 := gndx0 + xw0 + xw1 + sep01 + sep12 ; gndy2 := gndy0 ;
```

(\* Lay Vdd lines out \*)

```
lwire('metal',4.0,vddx0-xw0-5.0,vddy0) ;
```

```
x(vddx2) ;
```

```
nodelabel('Vdd!',vddx0-xw0-5.0,vddy0,'metal') ;
```

(\* Lay GND lines out \*)

```
wire(4.0,gndx0-3.0,gndy0) ;
```

```
x(gndx2) ;
```

```
nodelabel('GND!',gndx0-2.0,gndy0,'metal') ;
```

(\* Connect outputs of first clocked inverter to input of

```

second clocked inverter. *)
lwire('poly',polywidth,outx0,outty0);
y(outty0+4.0);
x(lnx2);
y(outty2);

```

```

(* Connect output of second clocked inverter to input of
first clocked inverter. *)
wire(polywidth,outx2,outty2);
y(outty0+14.0);
x(lnx0);
y(outty0);

```

```

(* Connect clock bar t line of first clocked inverter to clock
line of second clocked inverter. *)
wire(polywidth,clkbrx0,clkbrty0);
y(clkbrty0+4.0);

```

```

(* Draw a poly-metal contact *)
draw('rb',clkbrx0,clkbrty0+4.0);

```

```

(* Run a metal line over to the second clocked inverter *)
lwire('metal',metalwidth,clkbrx0,clkbrty0+4.0);
x(clkx2);

```

```

(* Draw a poly-metal contact to connect to top clock input of the
second clocked inverter. *)
draw('rb',clkx2,clkty2+4.0);

```

```

(* Connect to the poly-metal by poly *)
lwire('poly',polywidth,clkx2,clkty2+4.0);
y(clkty2);

```

```

(* Connect clock bar of the first inverter to clock bar b of
the second clocked inverter (using the bottom line). *)
wire(polywidth,clkx0,clkty0);
y(clkty0+9.0);

```

```

(* Draw a poly-metal contact to run metal to the second
clocked inverter. *)
draw('rb',clkx0,clkty0+9.0);

```

```

(* Run a metal line over to the second clocked inverter *)
lwire('metal',metalwidth,clkx0,clkty0+9.0);
x(clkbrx2);

```

```

(* Connect by a poly-metal contact to poly *)
draw('rb',clkbrx2,clkbrty2+9.0);
lwire('poly',polywidth,clkbrx2,clkbrty2+10.0);
y(clkbrty2);

```

```

(* Input signal *)
wire(polywidth,lnx0-5.0,inty0-23.0);
x(lnx0-2.0);

```

AD-A146 444

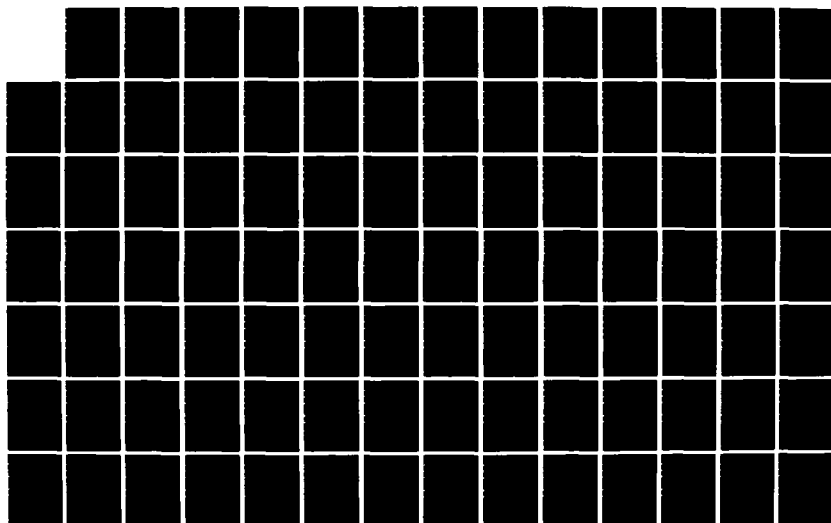
VLSI DESIGN TOOLS REFERENCE MANUAL RELEASE 20(U)  
WASHINGTON UNIV SEATTLE DEPT OF COMPUTER SCIENCE  
AUG 84 TR-84-88-07 NDA903-82-C-0424

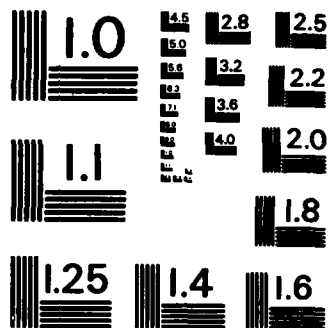
2/4

UNCLASSIFIED

F/G 9/5

NL





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A



```

(* Label the input line *)
nodeLabel('Signal_Left!',inx0-4.0,inty0-23.0,'poly') ;

(* Enable Left line *)
lwire('metal',metalwidth,inx0-8.0,clkby0-3.0) ;
x(clkbrx2) ;

(* Place a poly-metal contact to connect the Enable Left line
to clock of first inverter, clockbar of second, and
the input of basic inverter *)
draw('rb',clkx0,clkby0-3.0) ;
draw('rb',clkbrx2,clkbrby2-3.0) ;
draw('rb',inx1,inby1-3.0) ;

(* Connect the Enable line to the basic inverter and connect the
output of the inverter to the clock bar input of the
first clocked inverter and the clock input of the second
clocked inverter *)
lwire('poly',polywidth,inx1,inby1) ;
y(inby1-3.0) ;
wire(polywidth,clkx0,clkby0) ;
y(clkby0-3.0) ;
wire(polywidth,clkbrx2,clkbrby2-3.0) ;
y(clkbrby2-3.0) ;

(* Output of inverter *)
wire(polywidth,outx1,outby1) ;
y(outby1-10.0) ;

(* Poly-metal contacts for output of inverter, clock bar of
first clocked inverter, and clock line of second
clocked inverter *)
draw('rb',outx1,outby1-10.0) ;
draw('rb',clkbrx0,clkbrby0-10.0) ;
draw('rb',clkx2,clkby2-10.0) ;

(* Connect poly to these lines from the poly-metal contact *)
wire(polywidth,clkx2,clkby2-10.0) ;
y(clkby2) ;
wire(polywidth,clkbrx0,clkbrby0-10.0) ;
y(clkbrby0) ;

(* Run a metal strip connecting them all *)
lwire('metal',metalwidth,clkbrx0,clkbrby0-10.0)
x(clkx2) ;

(* Label the clock bar line *)
nodeLabel('EnLeft!',inx0-4.0,clkbrby0-3.0,'metal') ;

(* Output line - from basic inverter *)
lwire('poly',polywidth,outx2,outty2-23.0) ;
x(outx2+5.0) ;

(* Label the Output line *)

```

```
modelabel('Signal_Right',outx2+4.0,outty2-23.0,'poly');
```

```
enddef;
```

```
end; (** user **)
```

It would be useful, after compiling the above file as an exercise, to view the file on a GP-19 graphics terminal using the program `vic`. This will show the user how the design is actually appears when laid out by the above code.

# Placement and Routing Procedures

*Rick Hewitt*

UW-NW VLSI Consortium  
(Microtel Pacific Research)

## 1. Introduction

A forty-pin pad frame and a cell library based on the ISOCMOS process has been developed, along with a set of PLAP procedures to manipulate them. The library is intended to be used as a reference only. It is anticipated that an experienced designer would use the cells in this library as templates to design a library specifically suited for a particular chip.

Two sets of PLAP procedures are available for assisting in the layout of integrated circuits. The first set performs a semi-automatic placement and routing of cells to form larger cells in a hierarchical manner. The second set automatically places pad driver circuits next to their associated pads in the pad frame. With these procedures, the layout of an integrated circuit could proceed as follows:

- lay out low-level cells using either PLAP or CAESAR,
- combine these cells and cells from the library into larger cells using the placement and routing software,
- place pad drivers around the I/O pads in a pad frame,
- lay out programmable logic arrays (pla's) using TPLA,
- place the large cells, pla's and any other modules into the pad frame and interconnect them using CAESAR.

Examples of combining cells into larger cells and placing pad drivers around the I/O pads are given in the following PLAP programs:

`cells.p`  
`pads.p`

Both programs can be found under the directory `$UW_VLSI_TOOLS/src/examples/plap`.

## 2. The Cell Library

The library is located in

**\$UW\_VLSI\_TOOLS/lib/cellib/isocmos**

and it consists of the following cells:

<b>inv</b>	(inverter)
<b>nand2</b>	(2-input nand gate)
<b>nand3</b>	(3-input nand gate)
<b>nand4</b>	(4-input nand gate)
<b>nor2</b>	(2-input nor gate)
<b>nor3</b>	(3-input nor gate)
<b>nor4</b>	(4-input nor gate)
<b>xor2</b>	(2 input exclusive-or gate)
<b>clkinv</b>	(clocked inverter)
<b>sel2inv</b>	(2-input inverting selector)
<b>datch</b>	(d-type latch)
<b>datchr</b>	(d-type latch with reset)
<b>inpad</b>	(input pad)
<b>outpad</b>	(output pad)
<b>tripad</b>	(tristate pad)
<b>tripadm</b>	(mirror image of tristate driver pad)
<b>frame40</b>	(forty pin pad frame)
<b>vdd</b>	(contact to vdd)
<b>gnd</b>	(contact to ground)

Each cell has terminal points available which can be connected to terminal points on other cells. In general, each terminal is available at both the top and bottom of the cell. Each terminal point on the top of the cell has the suffix "t" and each terminal point on the bottom of the cell has the suffix "b". Thus, the terminal "in" has two points associated with it: "int", and "inb" which are identical electrically. These terminals are accessed automatically by the placement and routing software.

## 3. Placing and Routing Cells

The cells in the library are easily accessed using the PLAP functions and procedures available in the file:

**cells.psh**

which is in the directory **\$UW\_VLSI\_TOOLS/lib/psh**. This file contains the shell of a program which allows the use of a set of procedures and functions to support the placement of cells in a row and the interconnection of terminal points on these cells. This file is a PLAP file and once the commands are entered into it, it should be renamed so that the file name ends with ".p". The PLAP compiler is then called by typing the UNIX command:

**plap -ca -lsocells <file>**

Cells that are defined within the file are then available as PLAP symbols in the subdirectory "lib". Note that the flag **-lsocells** causes a plap library to be linked to the user program to perform the specified placement and routing functions. The **-ca** flag causes the cells to be defined in the CAESAR format.

For example, consider the following PLAP program that is typed into **cells.psh**.

```
var inv1, inv2, inv3 : SYMBOL;  
    net1, net2 : NET;  
  
defineCell( 'test' );  
  
    inv1 := placeCell( 'inv', normal );  
    inv2 := placeCell( 'inv', normal );  
    inv3 := placeCell( 'inv', normal );  
  
    net1 := startNet( inv1, 'inb' );  
    net2 := connectPoints( inv1, 'outt', inv2, 'int' );  
    net2 := connectPoints( inv2, 'outt', inv3, 'int' );  
    continueNet( net1, inv3, 'outh' );  
    exportNet( net1, west, 'ringOut' );  
  
endCell; { test }
```

This file defines a cell called "test" which consists of three inverters each of which has its output connected to the next inverter's input. The output of this "ring oscillator" is exported so that any cell which uses "test" as a subcell can connect to that point as a terminal. Note that once "test" is defined, only the net that is exported is available for connection. All other terminal points are hidden from cells at higher levels in the hierarchy.

There can be an arbitrary number of cells defined in each file, however, each cell definition must be surrounded by the commands:

```
defineCell( cellName );  
endCell;
```

```
cellName : string;
```

Cells are placed next to each other using the command:

```
sym := placeCell( cellName, orientation );
```

```
sym : SYMBOL;  
cellName : string;  
orientation : ( normal, mirrorx );
```

Cells are placed horizontally, from left to right in the order specified in the program. User-defined cells must adhere to the interface standard described in section 5 of this report. If these standards are used, power and ground lines abut and no design rules are introduced between adjacent cells or between channels when the interconnection is performed. The

returned value from this function (sym) is used to identify each instance of a particular cell and must be placed into a variable which has been defined to be of type SYMBOL. These values are used when specifying connections.

There are two alternative methods of specifying connections between terminal points within a cell. The first is convenient when only two points are to be connected:

```
net := connectPoints( sym1, ptName1, sym2, ptName2 );
```

```
net : NET;  
sym1, sym2 : SYMBOL;  
ptName1, ptName2 : string;
```

The value returned by this function must be stored in a variable that has been declared to be of type NET. The following commands can be used to specify an arbitrary number of terminal points connected together:

```
net := startNet( sym, ptName );  
continueNet( net, sym, ptName );
```

```
net : NET;  
sym : SYMBOL;  
ptName : string;
```

Once a net is started (either using startNet or connectPoints), that net can be continued an arbitrary number of times using an arbitrary number of calls to the procedure "continueNet".

No point within a cell is accessible to any other cell unless it is associated with a net which has been exported using the command:

```
exportNet( net, direction, netName );
```

```
net : NET;  
direction : (north, south, east, west);  
netName : string;
```

The terminal of an exported net will be extended to the specified edge. Often, a terminal point must be exported without being connected to any other terminal point in that cell. This is acceptable and would result in a "startNet" command followed by an "exportNet" command with no "continueNet" commands.

This is a preliminary version of the placement and routing software, consequently, only terminals on one side of a cell can be connected by a net. Currently, the only way to connect to points around a corner is to export the points in the proper direction and connect them at the next highest level in the hierarchy. Similarly, a net on the top of a cell can not be exported south, one on the left can not be exported east, one on the right can not be exported west and one on the bottom can not be exported north.

#### 4. The Pad Frame

A pad frame is available with a capacity of up to 38 user-defined pads plus power and ground. Pin 20 is reserved for ground and pin 40 for power. Each of the other pins can accommodate one of the following pad driver circuits:

- inpad (input pad)
- outpad (output pad)
- tripad (tri-state pad)

The placement of pad drivers can be done automatically in a fashion similar to placing cells. As with cells, there is a shell file which includes a set of procedures to support the automatic placement of pad drivers. This file is:

**pads.psh**

in the directory: **\$UW\_VLSI\_TOOLS/lib/psh**

The PLAP compiler is then called by typing the UNIX command:

**plap -ca -lsopads <file>**

As with the cells, the flag **-lsopads** causes a plap library to be linked to the user program to perform the specified pad placement.

The following PLAP program uses these procedures as an example:

```
defineChip( 'testChip' );  
  
  placePad( 'inpad', 1 );  
  placePad( 'outpad', 2 );  
  placePad( 'tripad', 3 );  
  
endChip; { testChip }
```

In general, each pad is placed by calling the procedure:

```
placePad( padType, pinNumber );  
  
padType : ('inpad', 'outpad', 'tripad');  
pinNumber : (1-19, 21-39);
```

The resulting plap cell, "testChip", will be available in the directory "lib".

## 5. Guidelines For Designing Custom Cells

This section describes the interface standard used by the cells in the library. If all custom cells adhere to this standard, then both custom and standard cells can be used indistinguishably without fear of introducing design rule errors between adjacent cells or between cells and the interconnection network. For the purpose of this report, the term "conducting material" means either metal, poly or diffusion (both n and p type).

1. Each cell must contain power and ground lines which are exactly four lambda wide. These lines are made of metal and run horizontally.
  - a) The power line may be at the top and must be labeled vdd. The ground line must be at the bottom and may be labeled gnd. These labels are for the circuit extraction program "mextra". **UNDER NO CIRCUMSTANCES SHOULD THE LABELS VDD! AND GND! BE USED.** These labels are defined once in the pad frame and will be used by "mextra" to ensure that power and ground are supplied to all cells.
  - b) The pitch between power and ground lines in a cell is arbitrary, however, all cells used in the definition of a cell must have identical pitch. The pitch in the standard cells is 40 lambda.
  - c) In a cell, the only conducting material which can be above the power line or below the ground line are the terminal points and interconnection channels. Violating this rule could introduce design rule violations between a cell and the interconnection network.
2. Terminal points are two lambda by two lambda poly squares located either above the power line or below the ground line.
  - a) The coordinates of the center of these squares must be specified using the Caesar :label command.
  - b) No two terminal points can be closer than six lambda in the x direction. That is, four lambda spacing must be maintained between the poly squares.
3. The origin of a symbol is arbitrary. However, the following four points must be defined for each symbol:

gndLeft,  
vddRight,  
lowerLeft,  
upperRight,

Each of these are defined according to the following rules. The placeCell command maintains a POINT variable called CURRENTORIGIN. Each cell is placed so that the point gndLeft is coincident with CURRENTORIGIN. After the current cell is placed, CURRENTORIGIN.x is then set equal to vddRight.x to be ready for the next cell. Implicit interconnection of cells can be achieved by allowing conducting material from adjacent cells to make electrical contact. This is achieved by choosing vddRight.x and gndLeft.x appropriately. It is recommended that all cells be capable of passing a design rule check without the aid of other cells. Therefore, it is not recommended to have a one-lambda wire on the right edge of one cell which abuts to a one-lambda cell on the left of the next. Instead, both cells should have full-size



wires on their respective edges and  $\text{vddRight.x}$  and  $\text{gndLeft.x}$  should be chosen so that there is overlap between the two cells. As a result, conducting material which is shared between adjacent cells can appear to the left of  $\text{gndLeft}$  and to the right of  $\text{vddRight}$ . However, all other conducting material, which is not intended to make electrical contact with adjacent cells must adhere to the following rules to avoid introducing design rule violations when cells are placed.

- a)  $\text{gndLeft.y}$  is defined to be the y coordinate of the center of the ground line.
- b)  $\text{gndLeft.x}$  is defined as follows: if the left most conducting material (not making contact with the previous cell) is metal and/or diffusion then  $\text{gndLeft.x}$  is the x coordinate of that conducting material. If the left most conducting material is poly then  $\text{gndLeft.x}$  is one lambda left of that left most poly. Under no circumstances should the x coordinate of a terminal point be within four lambda of  $\text{gndLeft.x}$ .
- c)  $\text{lowerLeft.x}$  is equal to  $\text{gndLeft.x}$  or the x coordinate of the left most conducting material, whichever is farthest left.
- d)  $\text{lowerLeft.y}$  is equal to the y coordinate of the lowest conducting material in the symbol.
- e)  $\text{vddRight.y}$  is defined to be the y coordinate of the center of the power line.
- f)  $\text{vddRight.x}$  is defined as follows: if the right most conducting material is poly then  $\text{vddRight.x}$  is one lambda to the right of that poly. If the right most conducting material is metal and/or diffusion then  $\text{vddRight.x}$  is two lambda to the right of that conducting material. By definition, no label x coordinate will be within two lambda of  $\text{vddRight.x}$ .
- g)  $\text{upperRight.x}$  is equal to  $\text{vddRight.x}$  or the x coordinate of the right most conducting material, whichever is farthest right.
- h)  $\text{upperRight.y}$  is defined to be the y coordinate of the top conducting material in the symbol.

Notice that the y rules are symmetric with respect to the top and bottom while the x rules are not. The asymmetry is due to the one lambda spacing rule between poly and diffusion but two lambda spacing between two pieces of poly or two pieces of metal.

- 4. The design rules state that there must be an eight lambda spacing between p-type diffusion and n-type diffusion. For devices with forty lambda spacing between rails, the following rules must be adhered to at the boundaries of cells to prevent inter-cell design rule violations. A boundary is defined to be within eight lambda from the left or right edge of a cell. If these rules are violated, an adjacent cell may have a design rule violation between n and p diffusions.
  - a) N-type diffusion at a boundary must not be drawn above sixteen lambda from  $\text{gndLeft.y}$

- b) P-type diffusion at a boundary must not be drawn below sixteen lambda from vddRight.y
- c) The above rules exclude ohmic contacts.

## 6. Command Summary

```
defineCell( str );  
sym := placeCell( str, orientation );  
net := connectPoints( sym, str, sym, str );  
net := startNet( sym, str );  
continueNet( net, sym, str );  
exportNet( net, direction, str );  
endCell;
```

```
defineChip( str );  
placePad( str, int );  
endChip;
```

```
str : string;  
sym : SYMBOL;  
net : NET;  
orientation : (normal, mirrorx);  
direction : (north, south, east, west);  
int : integer;
```

## 7. Files Used

\$UW\_VLSI\_TOOLS/lib/celllib/isocmos/\*

\$UW\_VLSI\_TOOLS/lib/psh/cells.psh  
\$UW\_VLSI\_TOOLS/lib/psh/pads.psh

\$UW\_VLSI\_TOOLS/src/examples/plap/cells/cells.p  
\$UW\_VLSI\_TOOLS/src/examples/plap/pads/pads.p

\$UW\_VLSI\_TOOLS/src/plap/isocells\*  
\$UW\_VLSI\_TOOLS/src/plap/isopads\*

# Standard Cell Library Guide

UW/NW VLSI Consortium  
University of Washington  
Seattle, WA 98195

## 1. Introduction

Various pre-generated cells are made available to the layout designer for use in custom designs laid out using **plap** or **caesar**. The various cells are to be used with the particular technology under which they are listed. The designer should avoid using the same names of these symbols when designing his own symbols.

### 1.1. nMOS Cell Library

These cells were provided courtesy of MOSIS, and as such correspond to MOSIS specifications for fabrication. They are current as of April 1984. For more information on the pads and padframes, refer to the MOSIS document *nmos.doc* included in the cells' source directory.

#### 1.1.1. Padframes

Pin 1 of these padframes is connected to the substrate, and therefore should not be used for a signal line.

- 28p23x34 - 28 pin frame with dimensions of 2300 by 3400 microns ;
- 28p46x34 - 28 pin frame with dimensions of 4600 by 3400 microns ;
- 40p46x34 - 40 pin frame with dimensions of 4600 by 3400 microns ;
- 40p46x68 - 40 pin frame with dimensions of 4600 by 6800 microns ;
- 40p69x68 - 40 pin frame with dimensions of 6900 by 6800 microns ;
- 64p46x68 - 64 pin frame with dimensions of 4600 by 6800 microns ;
- 64p69x68 - 64 pin frame with dimensions of 6900 by 6800 microns ;
- 64p79x92 - 64 pin frame with dimensions of 7900 by 9200 microns ;
- 84p69x68 - 84 pin frame with dimensions of 6900 by 6800 microns ;
- 84p79x92 - 84 pin frame with dimensions of 7900 by 9200 microns ;

#### 1.1.2. Pads

The names of some of these cells have been shortened due to system limits on the length of filenames. The MOSIS names are listed in parentheses with the renamed version ahead of them in boldface.

- PadVdd** - VDD pad ;
- PadGround** - GND pad ;
- PadIn** - Protected signal input pad ;
- PadOut** - Output pad ;
- PadClkO** (**PadClockedOut**) - Clocked output pad ;
- Pad3State** (**PadTriState**) - Tri-state input/output pad ;
- PadClkBar** (**PadClockBar**) - Two-phase non-overlapping clock-bar pad ;

## 1.2. CMOSPW Cell Library

These cells are intended for fabrication under the MOSIS 3 micron CMOS process "CBP2", and correspond to MOSIS specifications. They do not use second poly or metal, and were laid out using  $\lambda = 1$  micron. The pads and padframes are provided courtesy of Paul Bassett of the Massachusetts Institute of Technology. For a description of the differences between the three pad groups covered below, as well as a more complete description of the individual pad drivers and padframes, see Appendix A.

### 1.2.1. Padframes

Pin 1 of these padframes is connected to the substrate and should not be used for a signal line.

- 28p46x34 - 28 pin padframe with dimensions of 4600 by 3400 microns;
- 40p46x68 - 40 pin padframe with dimensions of 4600 by 6800 microns;
- 40p69x68 - 40 pin padframe with dimensions of 6900 by 6800 microns;
- 64p69x68 - 64 pin padframe with dimensions of 6900 by 6800 microns;
- 64p79x92 - 64 pin padframe with dimensions of 7900 by 9200 microns;
- 84p79x92 - 84 pin padframe with dimensions of 7900 by 9200 microns;

### 1.2.2. Group 1 Pads

These pads are the largest of the three groups of CMOSPW pads provided, measuring 300 by 640 microns.

- pad1out - Output pad;
- pad1out-ttl - TTL output pad;
- pad1ts - Tristate pad;
- pad1in - Input pad;
- pad1bin - Buffered input pad;
- pad1bin-ttl - Buffered TTL input pad;
- pad1gnd - Ground pad;
- pad1vdd - Vdd pad;
- pad1space - Spacer pad for padframes;
- pad1 - Complete group of the pad1 cells;

### 1.2.3. Group 2 Pads

This group has pads measuring 200 by 430 microns, and does not include any output pads.

- pad2in - Input pad;
- pad2bin - Buffered input pad;
- pad2bin-ttl - Buffered TTL input pad;
- pad2gnd - Ground pad;
- pad2vdd - Vdd pad;
- pad2space - Spacer pad for padframes;
- pad2 - Complete group of the pad2 cells;

### 1.2.4. Group 3 Pads

Group 3 contains an only unbuffered input pad, with a Vdd and Ground pad, all being 200 by 306 micron size.

- pad3in - Input pad;
- pad3gnd - Ground pad;
- pad3vdd - Vdd pad;
- pad3space - Spacer pad for padframes;
- pad3 - Complete group of the pad3 cells;

### 1.2.5. Standard Logic Cells

Refer to Appendix C for a complete description of each symbol as well as information on simulation and testing.

- inv - Basic inverter;
- nand2 - Two input nand gate;
- nand3 - Three input nand gate;
- nand4 - Four input nand gate;
- nor2 - Two input nor gate;
- nor3 - Three input nor gate;
- nor4 - Four input nor gate;
- xor2 - Two input exclusive-or gate;
- clkinv - Clocked inverter;
- sel2inv - Two input inverted selector;
- dlatch - D Type latch;
- dlatchr - D Type latch with reset;

### 1.3. ISOCMOS Cell Library

This is the 5 micron GTE ISOCMOS process. Refer to Appendix B for a more complete description of each symbol as well as information on simulation and testing. Also refer to the document Forty Pin Pad Frame and ISOCMOS Library.

- inv - Basic inverter;
- nand2 - Two input nand gate;
- nand3 - Three input nand gate;
- nand4 - Four input nand gate;
- nor2 - Two input nor gate;
- nor3 - Three input nor gate;
- nor4 - Four input nor gate;
- xor2 - Two input exclusive-or gate;
- clkinv - Clocked inverter;
- sel2inv - Two input inverted selector;
- dlatch - D Type latch;
- dlatchr - D Type latch with reset;
- inpad - Input pad;
- outpad - Output pad;
- tripad - Tristate pad;
- tripadm - Mirror image of tristate pad;
- frame40 - Forty pin padframe;
- vdd - Connection to vdd line;
- gnd - Connection to ground line;

## **Appendix A**

---

### **CMOSPW PADS AND PADFRAMES**

These pads are intended for fabrication under the MOSIS 3 micron CMOS process "CBP2". They do not use second poly or metal, and were laid out using  $\lambda = 1$  micron. They are provided courtesy of Paul Bassett of the Massachusetts Institute of Technology.

## 1. The Pad Types

The pads are divided into 3 groups. All of the pads in each group are compatible with the other members of its group but the groups are not compatible with each other due to power bus mismatches. Some of the pertinent properties of each of the groups are briefly discussed below followed by a brief discussion of each of the pad types. Members of different groups that have the same function are differentiated by a number in the name e.g. pad11a vs pad21a; since Group 1 has a complete set of pads, each type of pad will be discussed briefly only for that group.

## 2. Group One

This group is the only complete group and the pads in this group are also the largest pads. All of the pads are 300 x 640 microns. This size is dictated mostly by the size of driver transistors and the input buffer stages on the output pads; however, the pads have been expanded somewhat to fit the pad-to-pad spacings of the MOSIS standard padframes more closely. Two consequences of this are that the pads could be made slightly smaller if this is desired or the spacings of the driver transistors could be increased slightly more to provide some additional latchup protection. The input protection on all of the input pads consists of a well resistor, approximately 20 x 20 microns, followed p+-to-substrate and n+-to-well diodes providing additional clipping above Vdd and below Gnd, both diodes are approximately 15 x 15 microns. These pads have two Vdd buses and one Ground bus. The top Vdd bus is 60 microns wide and the bottom Vdd bus is 20 microns wide and has a 20 micron wide strip of n+ diffusion under it providing a guard ring to separate the pads from the internal circuitry, this guard ring is only broken where the inputs and outputs to the pads cross it. The Ground bus which runs through the middle of the pads is 74 microns wide. The input and output signal lines extend past the lower Vdd bus by 6 microns to allow connecting to them without design rule violations or modifications to the pads. Therefore, even though all of the pads have their lower left-hand corner at 0,0, the lower left-hand corner of the lower Vdd buses are at 0,6.

**pad1out** - output pad. While this is intended for driving principally capacitive loads such as other MOS devices, it can sink current for TTL. The signal is presented at point "DATA". It presents a small, though not minimal, load on this point. Experiments show that it can source or sink about 30ma, and has a delay of about 20ns into a very light load and 25ns into 50pf.

**pad1out-ttl** - TTL output pad. This pad is similar to the regular output pad except that it has an n-type pullup and the input buffer has been changed to drive the pullup and pulldown separately. This pad is experimental in that it has not ever been fabricated; in place it simulates correctly, it pulls HIGH to between 2.5 and 3 volts. How high it will pull in real operation is the major point in question. If it does pull high enough for TTL compatibility, it should be faster than the regular output pad.

**pad1ts** - bidirectional tristate pad. If point "OUT-ENAB" is set low, the pin is left to float, and whatever signal comes in from the outside appears at point "IN" (which is not buffered). If point "OUT-ENAB" is set high, the signal on point "OUT" is placed on the pin (and is also available on "IN"). This presents a fairly small, though not minimal, load on "OUT", but a moderately heavy load (sorry) on "OUT-ENAB".

**pad1ia** - unbuffered input pad. This has the "lightning arrestor" resistor and protective diodes, but no logic. The signal appears at point "DATA".

**pad1bia** - buffered input. It presents both the true data at the point labeled "DATA", and the inverted data at "-DATA". Both are driven by fairly strong buffers.

**pad1bia-ttl** - TTL input pad. This has input amplifiers designed to have a threshold near 1.5 volts for sensing the output of TTL chips. It presents both the true data at the point labeled "DATA", and the inverted data at "-DATA", though the latter's threshold is not offset as far as it should be. The output from "DATA" is fairly strong but the "-DATA" output is weak.

**pad1gnd, pad1vdd** - Vdd and Gnd pads. The appropriate voltages come out on 100 micron wide metal lines. The ground bus is broken in the Vdd pad and the lower Vdd bus is broken on the Gnd but the guard ring does continue under the ground line, without any contacts (obviously).



**pad1space** - spacer for pad frames. This cell is mainly meant for making it easy to fill in spaces between pads; all it contains is the three power and ground buses.

— - short guard ring. This cell is a 40 micron long piece of the 20 micron wide guard ring.

— - long guard ring. This cell is a 400 micron long piece of the guard ring.

**pad1** - the complete group. A cell containing an instance of every cell in the group.

### 3. Group Two

The pads in the second group are much smaller than those in the first group, 200 x 430 microns. This size is dictated by the size of the buffers on the buffered input pads. There are no output pads in this family. Each of the pads again has two Vdd buses, which are both 20 microns wide, and one Ground bus, which is 28 microns wide. The input protection is the same as for the Group 1 input pads and there is also a 20 micron wide guard ring under the lower Vdd bus.

### 4. Group Three

This group actually consists of just an unbuffered input pad so the size of the other pads, Vdd etc., is dictated by this pad and is 200 x 306 microns. This pad has a 30 wide upper Vdd bus, a 10 micron wide Ground bus and a 20 micron wide lower Vdd bus with the guard ring under it like the other pads. The input protection on this pad is the same as on the others.

### 5. MOSIS Standard Pad Frames

Pad frames have been developed for some of the MOSIS standard pad frames. The frames contain the indicated number of pads, all of which are initially unbuffered input pads. The pads are arranged to meet the MOSIS specifications and where necessary, the **pad1space** instance has been used to fill in the buses and the guard ring in between the pads. Also, each of the pad frames has instantiated in the middle of it a set of the available pads. Since the output pads are fairly large, not all of the padframe spacing would allow using all of the allotted pins, only those frames that allow a full complement of pads have been implemented; the available frames are:

FRAME NAME	DIE SIZE (MICRONS)	INTERIOR PROJECT SIZE	PINS/PACKAGE	PIN ROW SPACING
28p46x34	4600 x 3400	3320 x 2120	28 DIP	0.6"
40p46x68	4600 x 6800	3320 x 5520	40 DIP	0.6"
40p69x68	6900 x 6800	5620 x 5520	40 DIP	0.6"
64p69x68	6900 x 6800	5620 x 5520	64 DIP	0.9"
64p79x92	7900 x 9200	6620 x 7920	64 DIP	0.9"
84p79x92	7900 x 9200	6620 x 7920	84 PGA	-

## **Appendix B**

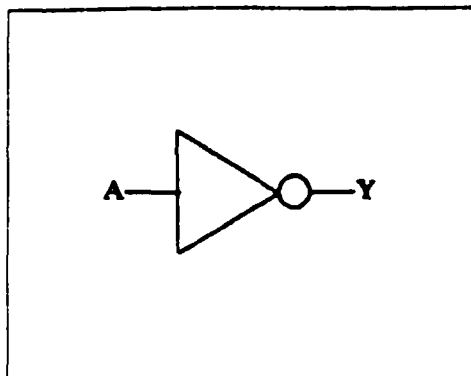
---

### **ISOCMOS CELL DESCRIPTIONS**

Herein is contained a description of most of the standard cells available in the ISOCMOS cell library. At the end of this appendix is located a writeup on simulation conditions.

## ISOCMOS LIBRARY CELL INV

## INVERTER



Truth Table

A	Y
0	1
1	0

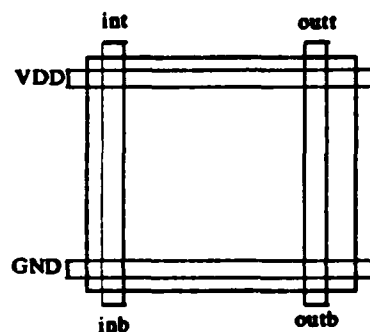
$$Y = \bar{A}$$

Nearest Functional Equivalent:

CMOS 4049, 4069  
TTL 7404

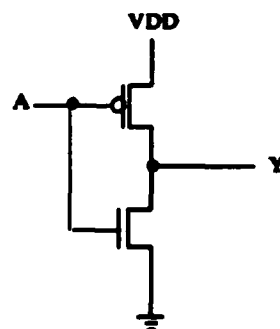
Nodes	A	Y								
Input Load	1	-								

Block Diagram of I/O pins

Cell Width: 16  $\lambda$ AC Characteristics  $V_{DD} = 5V$   
Input Transition Time = 5ns

Parameter		Fan Out Load = 1	Fan Out Load = 10
		Typ	Typ
Propagation delay (high to low)	$t_{PHL}$	1.0	4.5
Propagation delay (low to high)	$t_{PLH}$	2.0	5.5
Output fall time	$t_{THL}$	2.5	8.5
Output rise time	$t_{TLH}$	3.0	12.0

Circuit Schematic Diagram



## NOTES:

- 1) rail separation: 40  $\lambda$
- 2) bounding box (x x y): 16 $\lambda$  x 44 $\lambda$
- 3) W/L of p-channel transistor: 8/2
- 4) W/L of n-channel transistor: 4/2

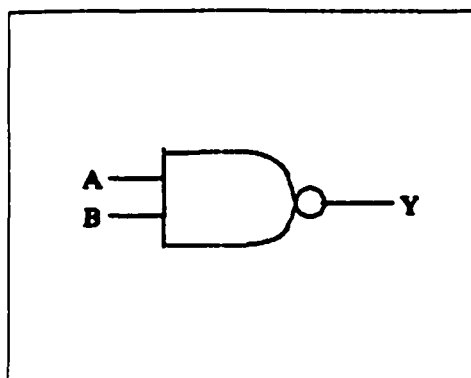
Table values from SPICE simulation

CAESAR file: \$UW\_VLSI\_TOOLS/src/cellib/isocmos/inv.ca

DB files: \$UW\_VLSI\_TOOLS/src/cellib/isocmos/(inv.sym, inv.att)

# ISOCMOS LIBRARY CELL NAND2

## 2-INPUT NAND



Truth Table

A	B	Y
0	X	1
X	0	1
1	1	0

$$Y = \overline{AB}$$

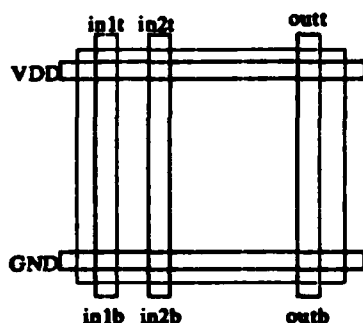
Nearest Functional Equivalent:

CMOS 4011  
TTL 7400

Nodes	A	B	Y						
Input Load	1	1	-						

Block Diagram of I/O pins

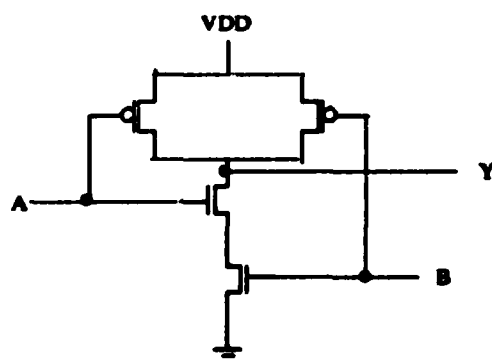
Cell Width: 24  $\lambda$



AC Characteristics  $V_{DD} = 5V$   
Input Transition Time = 5ns

Parameter		Fan Out Load = 1	Fan Out Load = 10
		Typ	Typ
Propagation delay (high to low)	$t_{PHL}$	2.0	6.5
Propagation delay (low to high)	$t_{PLH}$	2.5	10.0
Output fall time	$t_{THL}$	2.5	12.0
Output rise time	$t_{TLH}$	4.5	20.0

Circuit Schematic Diagram



### NOTES:

- 1) rail separation: 40  $\lambda$
- 2) bounding box (x  $\times$  y): 24  $\lambda \times$  44  $\lambda$
- 3) W/L of all transistors: 4/2

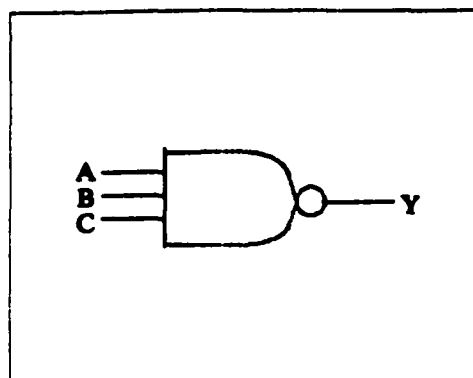
Table values from SPICE simulation

CAESAR file: \$UW\_VLSI\_TOOLS/src/cellib/isocmos/nand2.ca

DB files: \$UW\_VLSI\_TOOLS/src/cellib/isocmos/(nand2.sym, nand2.att)

# ISOCMOS LIBRARY CELL NAND3

# 3-INPUT NAND



Truth Table

A	B	C	Y
0	X	X	1
X	0	X	1
X	X	0	1
1	1	1	0

$$Y = \overline{ABC}$$

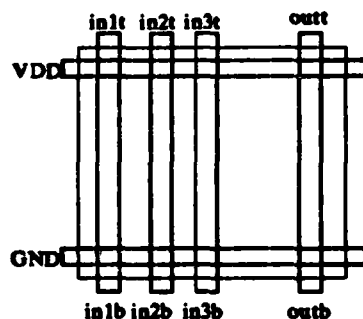
Nearest Functional Equivalent:

CMOS 4023  
TTL 7410

Nodes	A	B	C	Y						
Input Load	1	1	1	-						

Block Diagram of I/O pins

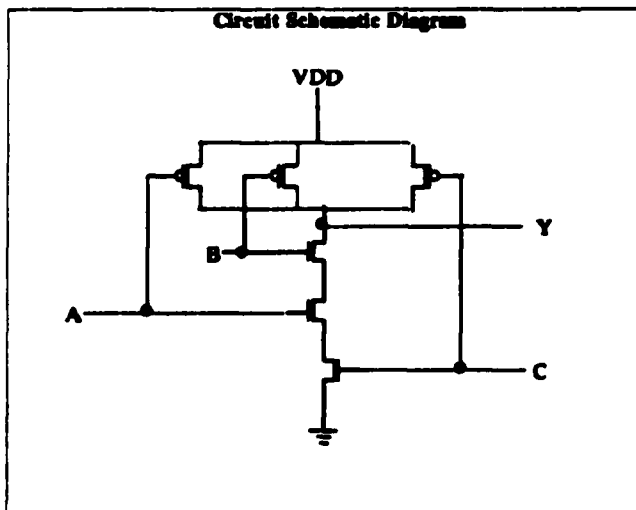
Cell Width: 32  $\lambda$



AC Characteristics  $V_{DD} = 5V$   
Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	$t_{PHL}$	2.5	8.5
Propagation delay (low to high)	$t_{PLH}$	3.0	11.5
Output fall time	$t_{FHL}$	3.5	17.0
Output rise time	$t_{FLH}$	4.5	21.5

Circuit Schematic Diagram



## NOTES:

- 1) rail separation: 40  $\lambda$
- 2) bounding box (x  $\times$  y): 32 $\lambda$   $\times$  44 $\lambda$
- 3) W/L of all transistors: 4/2

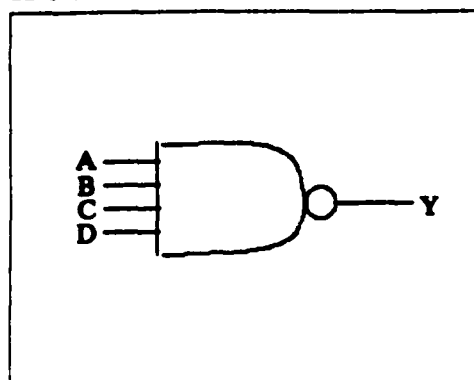
Table values from SPICE simulation

CAESAR file: \$UW\_VLSI\_TOOLS/src/cellib/isocmos/nand3.ca

DB files: \$UW\_VLSI\_TOOLS/src/cellib/isocmos/(nand3.sym, nand3.att)

# ISOCMOS LIBRARY CELL NAND4

# 4-INPUT NAND



Truth Table

A	B	C	D	Y
0	X	X	X	1
X	0	X	X	1
X	X	0	X	1
X	X	X	0	1
1	1	1	1	0

$$Y = \overline{ABCD}$$

Nearest Functional Equivalent:

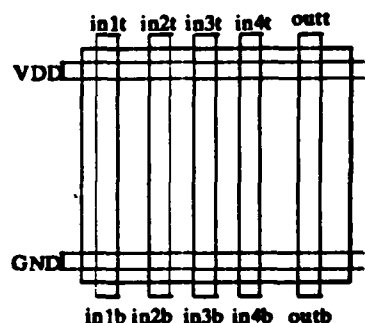
CMOS 4012

TTL 7420

Nodes	A	B	C	D	Y				
Input Load	1	1	1	1	-				

Block Diagram of I/O pins

Cell Width: 40  $\lambda$

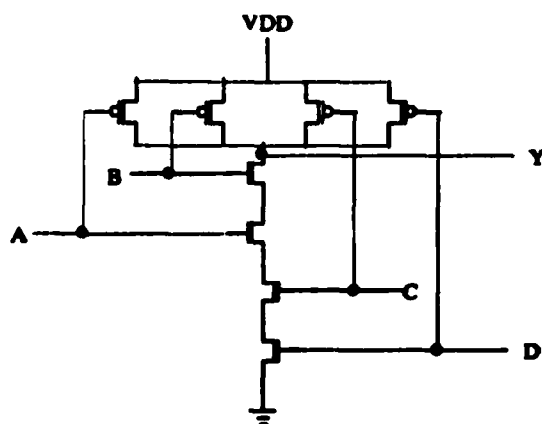


AC Characteristics  $V_{DD} = 5V$

Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	$t_{PHL}$	3.5	11.5
Propagation delay (low to high)	$t_{PLH}$	3.0	8.0
Output fall time	$t_{FHL}$	4.5	23.0
Output rise time	$t_{FLH}$	4.5	19.0

Circuit Schematic Diagram



## NOTES:

- 1) rail separation: 40  $\lambda$
- 2) bounding box (x  $\times$  y): 40 $\lambda$   $\times$  44 $\lambda$
- 3) W/L of all p-channel transistors: 16/4
- 4) W/L of all n-channel transistors: 8/4

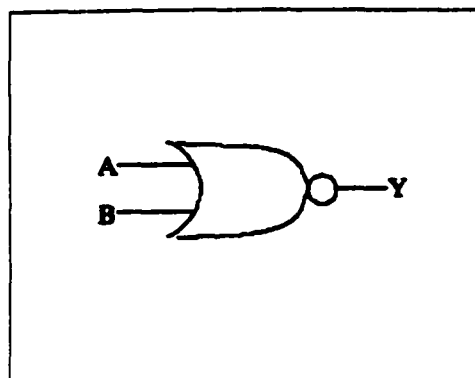
Table values from SPICE simulation

CAESAR file: \$UW\_VLSI\_TOOLS/src/cellib/isocmos/nand4.ca

DB files: \$UW\_VLSI\_TOOLS/src/cellib/isocmos/(nand4.sym, nand4.att)

# ISOCMOS LIBRARY CELL NOR2

## 2-INPUT NOR



Truth Table

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

$$Y = \overline{A + B}$$

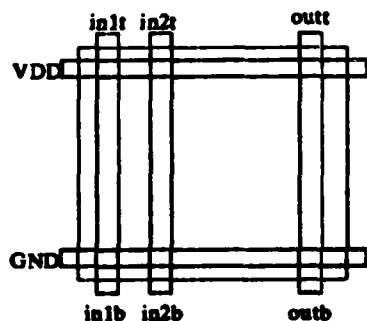
Nearest Functional Equivalent:

CMOS 4001  
TTL 7402

Nodes	A	B	Y							
Input Load	1	1	-							

Block Diagram of I/O pins

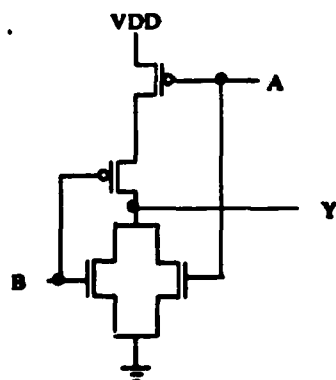
Cell Width: 24  $\lambda$



AC Characteristics  $V_{DD} = 5V$   
Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	$t_{PHL}$	1.0	2.5
Propagation delay (low to high)	$t_{PLH}$	4.5	21.5
Output fall time	$t_{THL}$	1.5	4.5
Output rise time	$t_{TLH}$	7.5	44.5

Circuit Schematic Diagram



### NOTES:

- 1) rail separation: 40  $\lambda$
- 2) bounding box (x x y): 24 $\lambda$  x 44 $\lambda$
- 3) W/L of all transistors: 4/2

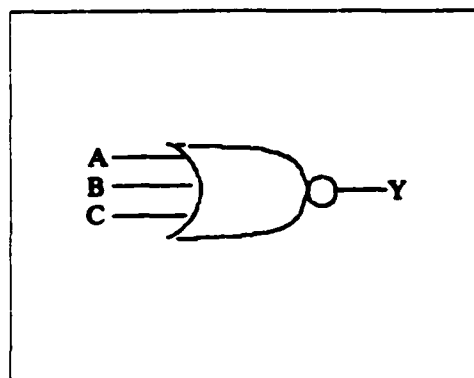
Table values from SPICE simulation

CAESAR file: \$UW\_VLSI\_TOOLS/src/cellib/isocmos/nor2.ca

DB files: \$UW\_VLSI\_TOOLS/src/cellib/isocmos/(nor2.sym, nor2.att)

# ISOCMOS LIBRARY CELL NOR3

# 3-INPUT NOR



Truth Table

A	B	C	Y
0	0	0	1
x	x	1	0
x	1	x	0
1	x	x	0

$$Y = \overline{A+B+C}$$

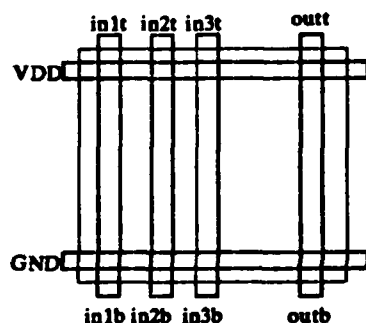
Nearest Functional Equivalent:

CMOS 4000  
TTL 7427

Nodes	A	B	C	Y					
Input Load	1	1	1	-					

Block Diagram of I/O pins

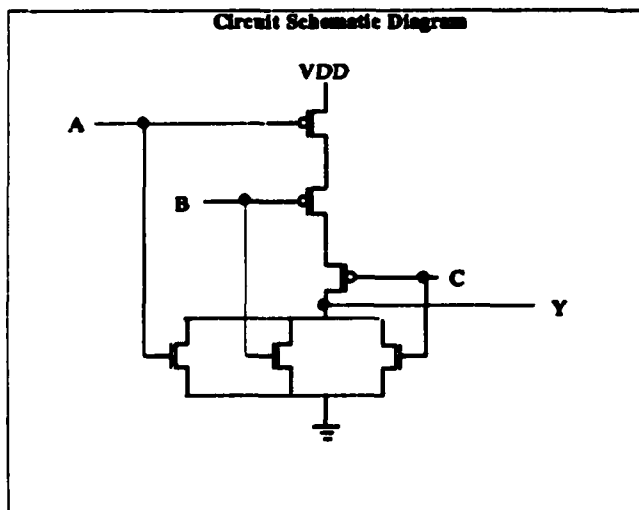
Cell Width: 32  $\lambda$



AC Characteristics  $V_{DD} = 5V$   
Input Transition Time = 5ns

Parameter		Fan Out Load = 1	Fan Out Load = 10
		Typ	Typ
Propagation delay (high to low)	$t_{PHL}$	15	50
Propagation delay (low to high)	$t_{PLH}$	70	340
Output fall time	$t_{THL}$	10	80
Output rise time	$t_{TLH}$	120	705

Circuit Schematic Diagram



## NOTES:

- 1) rail separation: 40  $\lambda$
- 2) bounding box (x  $\times$  y): 32 $\lambda$   $\times$  44 $\lambda$
- 3) W/L of all transistors: 4/2

Table values from SPICE simulation

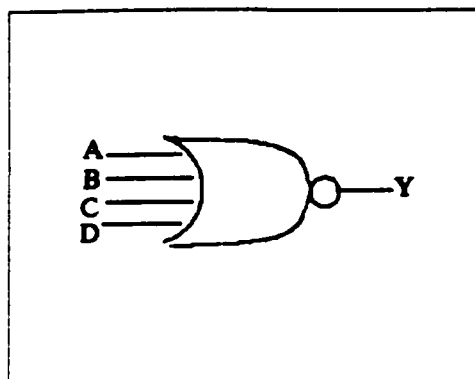
CAESAR file: \$UW\_VLSI\_TOOLS/src/cellib/isocmos/nor3.ca

DB files: \$UW\_VLSI\_TOOLS/src/cellib/isocmos/(nor3.sym, nor3.att)



# ISOCMOS LIBRARY CELL NOR4

## 4-INPUT NOR



Truth Table

A	B	C	D	Y
0	0	0	0	1
x	x	x	1	0
x	x	1	x	0
x	1	x	x	0
1	x	x	x	0

$$Y = \overline{A+B+C+D}$$

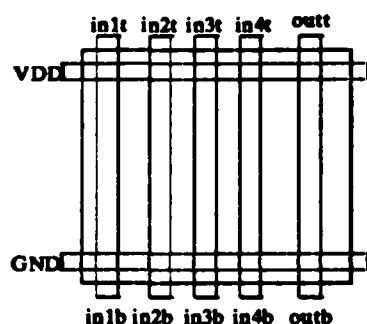
Nearest Functional Equivalent:

CMOS 4002  
TTL 7425

Nodes	A	B	C	D	Y					
Input Load	1	1	1	1	-					

Block Diagram of I/O pins

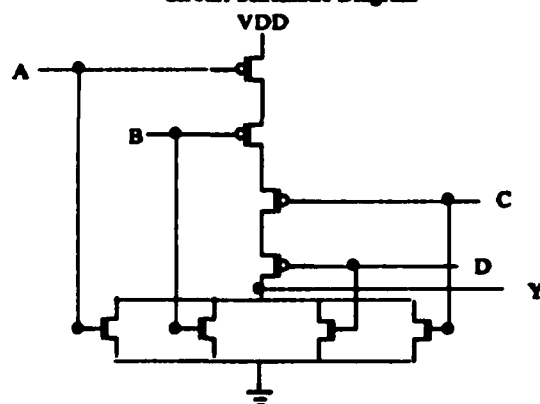
Cell Width: 40  $\lambda$



AC Characteristics  $V_{DD} = 5V$   
Input Transition Time = 5ns

Parameter		Fan Out Load = 1	Fan Out Load = 10
		Typ	Typ
Propagation delay (high to low)	$t_{PHL}$	0.5	4.5
Propagation delay (low to high)	$t_{PLH}$	10.0	48.5
Output fall time	$t_{FHL}$	1.0	8.0
Output rise time	$t_{TLH}$	17.5	95.0

Circuit Schematic Diagram



### NOTES:

- 1) rail separation: 40  $\lambda$
- 2) bounding box (x y): 40 $\lambda$  x 44 $\lambda$
- 3) W/L of all transistors: 4/2

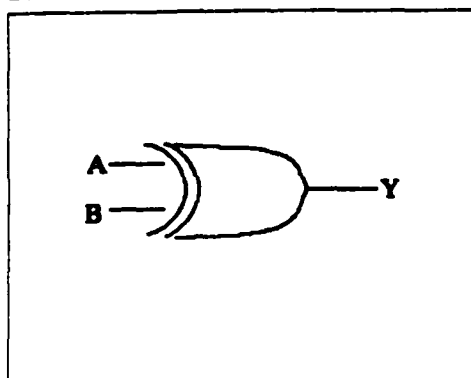
Table values from SPICE simulation

CAESAR file: \$UW\_VLSI\_TOOLS/src/cellib/isocmos/nor4.ca

DB files: \$UW\_VLSI\_TOOLS/src/cellib/isocmos/(nor4.sym, nor4.att)

# ISOCMOS LIBRARY CELL XOR2

## 2-INPUT XOR



Truth Table

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

$$Y = A \oplus B$$

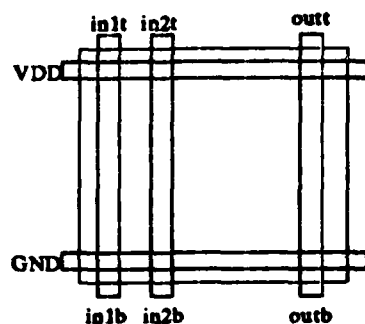
Nearest Functional Equivalent:

CMOS 4030  
TTL 7486

Nodes	A	B	Y						
Input Load	2	2	-						

Mock Diagram of I/O pins

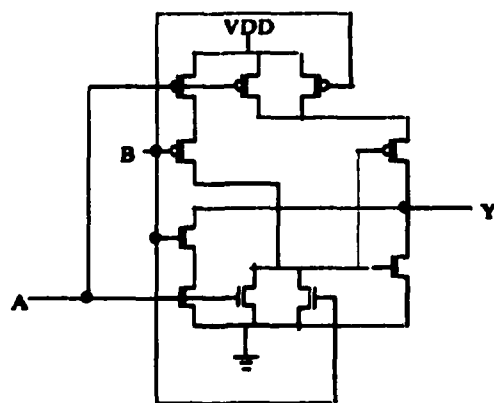
Cell Width: 32  $\lambda$



AC Characteristics  $V_{DD} = 5V$   
Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	$t_{PHL}$	2.5	6.0
Propagation delay (low to high)	$t_{PLH}$	5.0	21
Output fall time	$t_{THL}$	1.5	12.5
Output rise time	$t_{TLH}$	8.0	45.0

Circuit Schematic Diagram



### NOTES:

- 1) rail separation: 40  $\lambda$
- 2) bounding box (x  $\times$  y): 32 $\lambda$   $\times$  44 $\lambda$
- 3) W/L of all transistors: 4/2

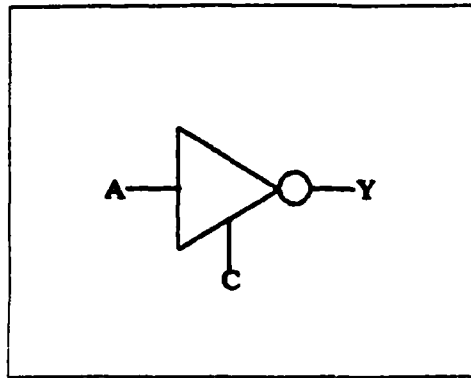
Table values from SPICE simulation

CAESAR file: \$UW\_VLSI\_TOOLS/src/cellib/isocmos/xor2.ca

DB files: \$UW\_VLSI\_TOOLS/src/cellib/isocmos/(xor2.sym, xor2.att)

## ISOCMOS LIBRARY CELL CLKINV

## CLOCKED INVERTER



Truth Table

C	C-	A	Y
0	1	0	Z
0	1	1	Z
1	0	0	1
1	0	1	0

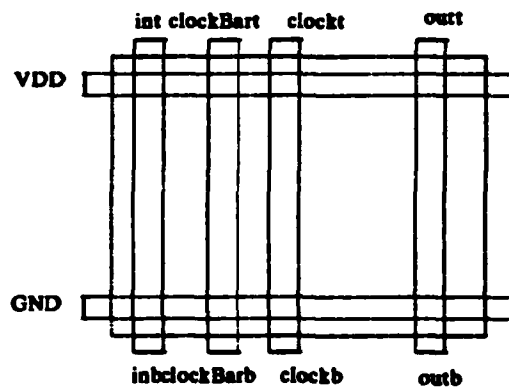
$$Y = \bar{A}$$

Nearest Functional  
Equivalent:

CMOS  
TTL

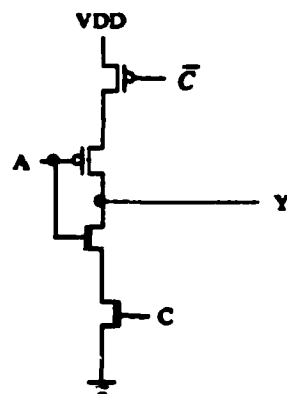
Nodes	C	C-	A	Y						
Input Load	33	.67	1	-						

Block Diagram of I/O pins

Cell Width: 33  $\lambda$ AC Characteristics  $V_{DD} = 5V$   
Input Transition Time = 5ns

Parameter		Fan Out Load = 1	Fan Out Load = 10
		Typ	Typ
Propagation delay (high to low)	$t_{PHL}$	2.0	6.0
Propagation delay (low to high)	$t_{PLH}$	3.0	11.0
Output fall time	$t_{THL}$	3.0	12.5
Output rise time	$t_{TLH}$	4.5	23.0

Circuit Schematic Diagram



## NOTES:

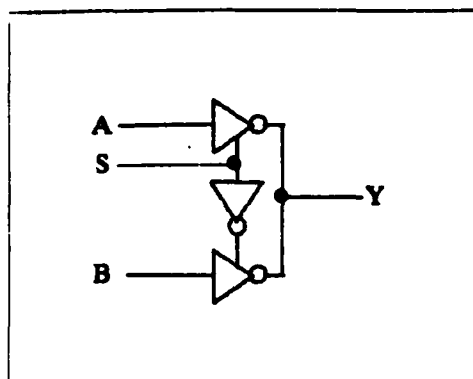
- 1) rail separation: 40  $\lambda$
- 2) bounding box (x y): 33 $\lambda$  x 44 $\lambda$
- 3) W/L of all p-channel transistors: 8/2
- 4) W/L of all n-channel transistors: 4/2

Table values from SPICE simulation

CAESAR file: \$UW\_VLSI\_TOOLS/src/cellib/isocmos/clkinv.ca  
DB files: \$UW\_VLSI\_TOOLS/src/cellib/isocmos/(clkinv.sym, clkinv.att)

## ISOCMOS LIBRARY CELL SEL2INV

## 2-INPUT INVERTED SELECTOR



Truth Table

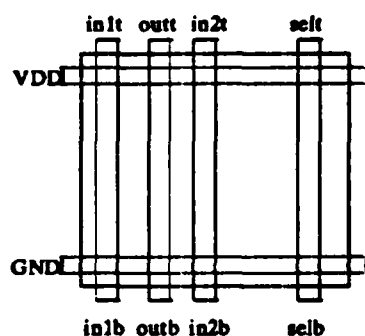
S	A	B	Y
1	1	X	0
1	0	X	1
0	X	1	0
0	X	0	1

Nearest Functional  
Equivalent:

CMOS  
TTL 74157

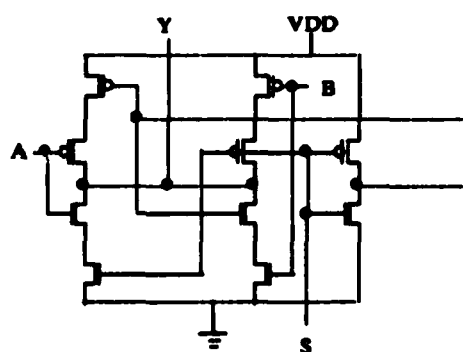
Nodes	S	A	B	Y					
Input Load	2	1	1	-					

Block Diagram of I/O pins

Cell Width: 54  $\lambda$ AC Characteristics  $V_{DD} = 5V$   
Input Transition Time = 5ns

Parameter		Fan Out Load = 1	Fan Out Load = 10
		Typ	Typ
Propagation delay (high to low)	$t_{PHL}$	4.5	11.5
Propagation delay (low to high)	$t_{PLH}$	5.0	17.5
Output fall time	$t_{THL}$	4.9	19.5
Output rise time	$t_{TLH}$	7.1	36.5

Circuit Schematic Diagram



## NOTES:

- 1) rail separation: 40  $\lambda$
- 2) bounding box (x  $\times$  y): 54  $\lambda \times$  44  $\lambda$
- 3) W/L of all p-channel transistors: 8/2
- 4) W/L of all n-channel transistors: 4/2

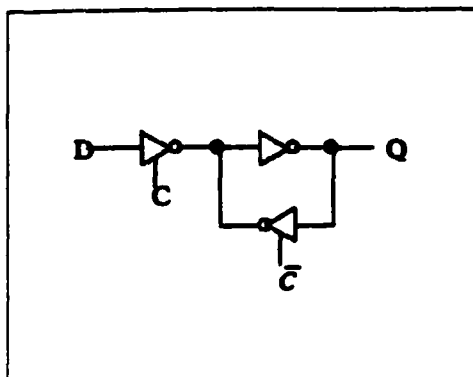
Table values from SPICE simulation

CAESAR file: \$UW\_VLSI\_TOOLS/src/cellib/isocmos/sel2inv.ca

DB files: \$UW\_VLSI\_TOOLS/src/cellib/isocmos/(sel2inv.sym, sel2inv.att)

# ISOCMOS LIBRARY CELL DLATCH

# D-LATCH



Truth Table

C	C-	D	Q
1	0	1	1
1	0	0	0
0	1	X	Q

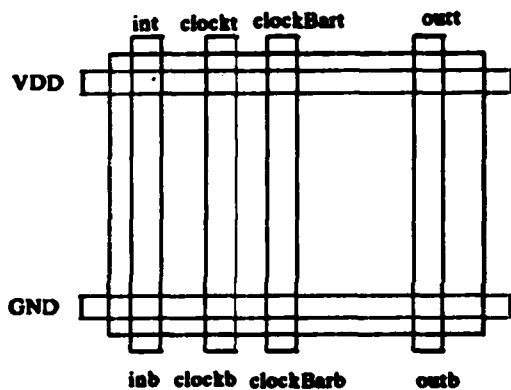
Nearest Functional Equivalent:

CMOS  
TTL 74LS77

Nodes	C	C-	D	Q						
Input Load	1	1	1	-						

Block Diagram of I/O pins

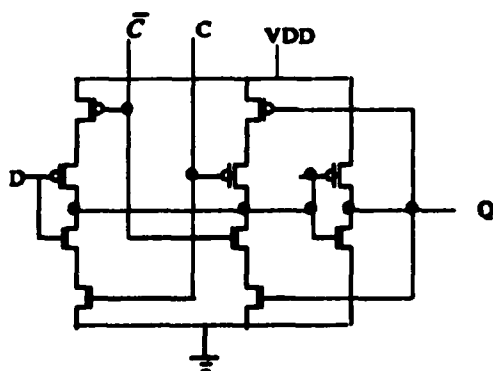
Cell Width: 57  $\lambda$



AC Characteristics  $V_{DD} = 5V$   
Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	$t_{PHL}$	5.0	8.0
Propagation delay (low to high)	$t_{PLH}$	4.0	7.5
Output fall time	$t_{THL}$	3.5	8.5
Output rise time	$t_{TLH}$	3.5	12.0

Circuit Schematic Diagram



## NOTES:

- 1) rail separation: 40  $\lambda$
- 2) bounding box (x x y): 57  $\lambda$  x 44  $\lambda$
- 3) W/L of all p-channel transistors: 8/2
- 4) W/L of all n-channel transistors: 4/2

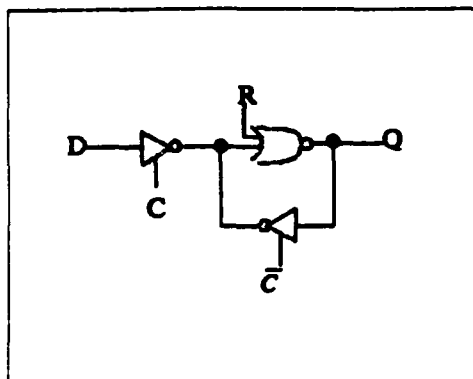
Table values from SPICE simulation

CAESAR file: \$UW\_VLSI\_TOOLS/src/cellib/isocmos/dlatch.ca

DB files: \$UW\_VLSI\_TOOLS/src/cellib/isocmos/(dlatch.sym, dlatch.att)

# ISOCMOS LIBRARY CELL DLATCHR

# D-LATCH WITH RESET



Truth Table

C	C-	R	D	Q
1	0	0	D	D
0	1	0	X	Q
X	X	1	X	0

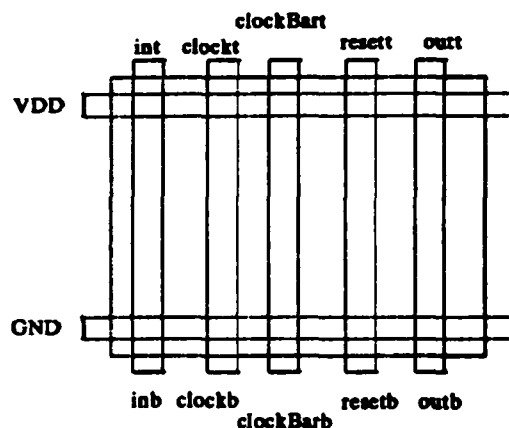
Nearest Functional Equivalent:

CMOS  
TTL

Nodes	C	C-	R	D	Q				
Input Load	1	1	1	1	-				

Block Diagram of I/O pins

Cell Width: 63  $\lambda$

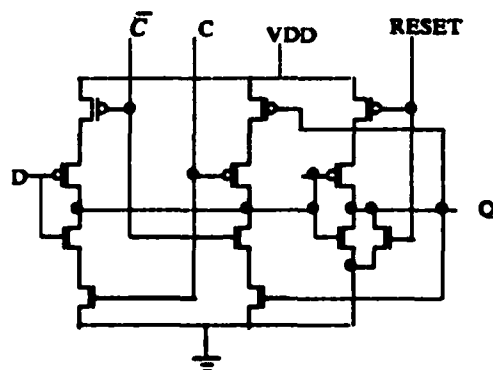


AC Characteristics  $V_{DD} = 5V$

Input Transition Time = 5ns

Parameter		Fan Out Load = 1	Fan Out Load = 10
		Typ	Typ
Propagation delay (high to low)	$t_{pHL}$	4.5	8.0
Propagation delay (low to high)	$t_{pLH}$	5.0	13.5
Output fall time	$t_{THL}$	3.0	8.5
Output rise time	$t_{TLH}$	6.5	25.0

Circuit Schematic Diagram



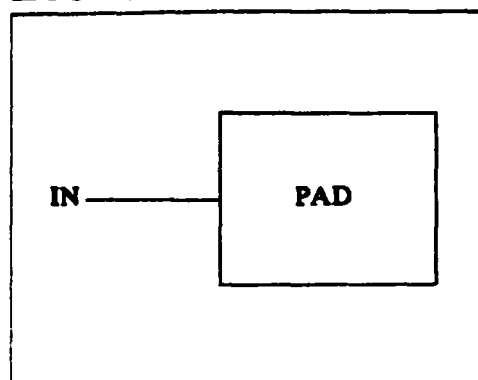
## NOTES:

- 1) rail separation: 40  $\lambda$
- 2) bounding box (x x y): 63 $\lambda$  x 44 $\lambda$
- 3) W/L of all p-channel transistors: 8/2
- 4) W/L of all n-channel transistors: 4/2
- 5) Reset propagation delay: 1.5 4.5
- 6) Reset fall time: 3.0 9.0

Table values from SPICE simulation

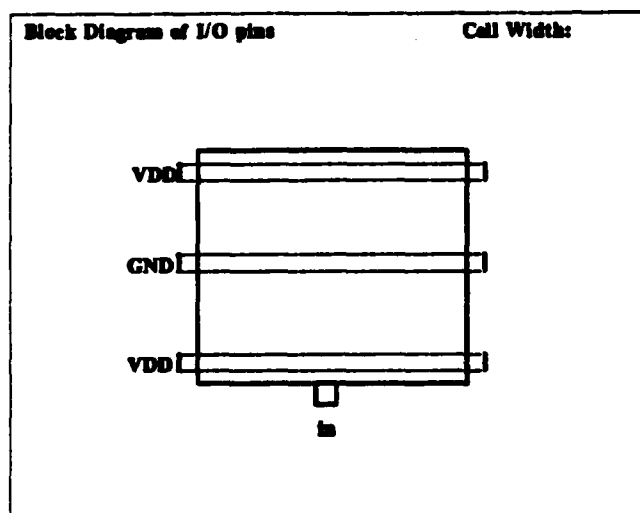
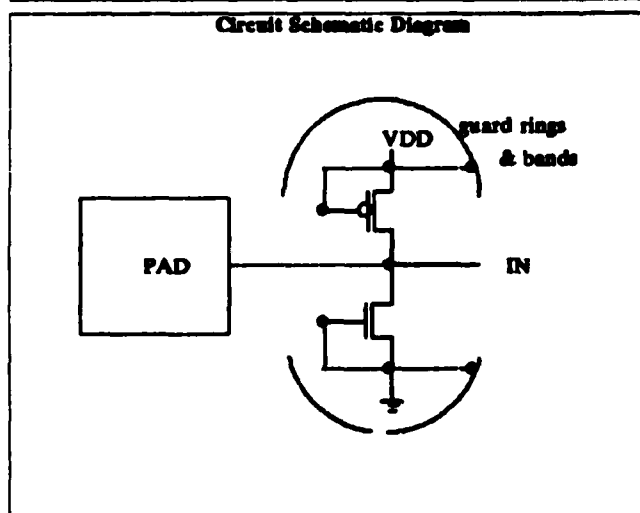
CAESAR file: \$UW\_VLSI\_TOOLS/src/uw-vlsi/isocmos/dlatchr.ca

DB files: \$UW\_VLSI\_TOOLS/src/cellib/isocmos/(dlatchr.sym, dlatchr.att)

**INPAD**

### Truth Table

**Nearest Functional Equivalent:**

CMOS  
TTL[illegible][illegible]

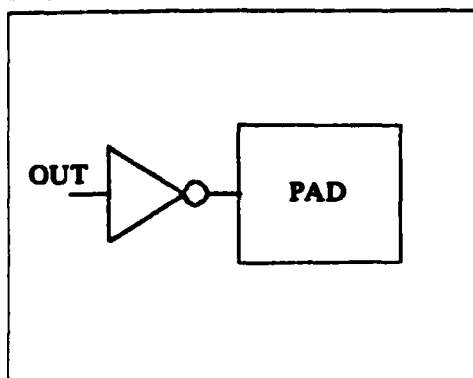
**NOTES:**

**CAESAR file:** \$UW\_VLSI\_TOOLS/src/cellib/isocmos/inpad.ca

**DB files:** \$UW\_VLSI\_TOOLS/src/cellib/isocmos/(inpad.sym, inpad.att)

# ISOCMOS LIBRARY CELL OUTPAD

# OUTPAD



Truth Table

OUT	PAD
0	1
1	0

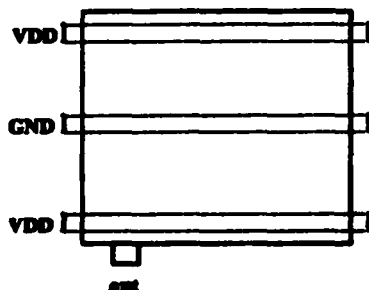
Nearest Functional Equivalent:

CMOS  
TTL 7404

Nodes	OUT	PAD							
Input Load	1	-							

Block Diagram of I/O pins

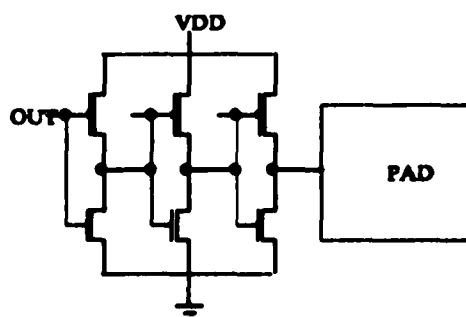
Cell Width:



AC Characteristics  $V_{DD} = 5V$   
Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	$t_{PHL}$	8.0	14.5
Propagation delay (low to high)	$t_{PLH}$	8.6	16.5
Output fall time	$t_{THL}$	3.5	13.0
Output rise time	$t_{TLH}$	4.0	24.0

Circuit Schematic Diagram



## NOTES:

Table values from SPICE simulation

Load=1 means 5 pF

Load=10 means 50 pF

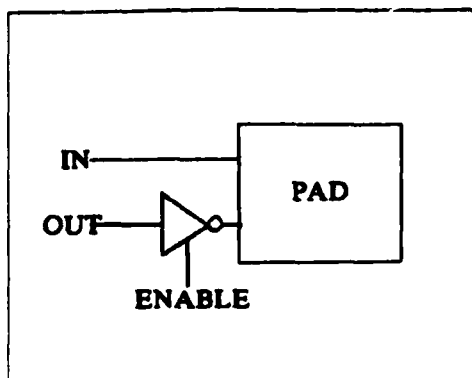
CAESAR file: \$UW\_VLSI\_TOOLS/src/cellib/isocmos/outpad.ca

DB files: \$UW\_VLSI\_TOOLS/src/cellib/isocmos/(outpad.sym, outpad.att)



# ISOCMOS LIBRARY CELL TRIPAD

# TRIPAD



Truth Table

EN	OUT	PAD
1	0	1
1	1	0
0	X	Z

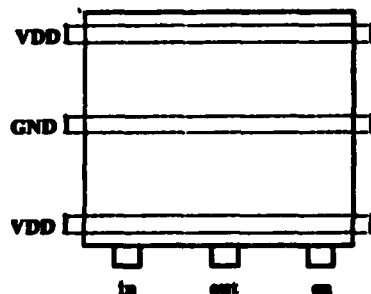
Nearest Functional Equivalent:

CMOS  
TTL

Nodes	EN	OUT	PAD						
Input Load	2	2	-						

Block Diagram of I/O pins

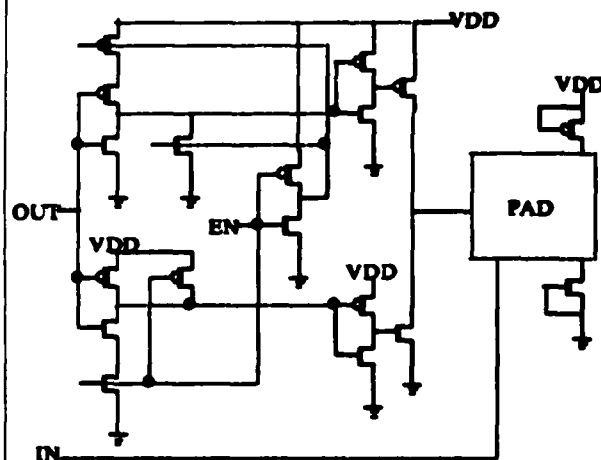
Cell Width:



AC Characteristics  $V_{DD} = 5V$   
Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	$t_{PHL}$	5.6	12.0
Propagation delay (low to high)	$t_{PLH}$	8.5	20.5
Enable time to high level	$t_{PZH}$	10.6	22.5
Enable time to low level	$t_{PZL}$	-1.6	5.5
Disable time from high level	$t_{PHZ}$	14.0	32.0
Disable time to low level	$t_{PLZ}$	-1.6	0.0
Output fall time	$t_{THL}$	3.1	12.5
Output rise time	$t_{TLH}$	5.5	37.5

Circuit Schematic Diagram



## NOTES:

Table values from SPICE simulation

Load = 1 means 5 pF

Load = 10 means 50 pF

adjacent tripads must be mirrored to avoid design-rule errors

CAESAR file: \$UW\_VLSI\_TOOLS/src/celllib/isocmos/tripad.ca

DB files: \$UW\_VLSI\_TOOLS/src/celllib/isocmos/(tripad.sym, tripad.att)

## Simulation Conditions

All standard cells were simulated by *spice* (a circuit simulator program). The circuits were generated from the layout artwork by first creating CIF (Caltech Intermediate Form) files from the layout. The circuit extraction program *meatra* was then used to extract the circuit from the CIF file. Following are some electrical data of the timing simulations.

Temperature parameter was set to 27°C

Control signal transition time was 5 ns

VDD = 5V      GND = 0V

Output fall time (\$t\_{sub THL}\$) was measured from 90% (4.5V) point to 10% point (0.5V) of the output signal.

Output rise time (\$t\_{sub TLH}\$) was measured from 10% (0.5V) point to 90% point (4.5V) of the output signal.

Propagation delay (\$t\_{sub pHL}\$ or \$t\_{sub pLH}\$) was measured from the control signal 50% point (2.5V) to the output signal 50% point (2.5V).

Output disable time from high level (\$t\_{sub pHZ}\$) was measured as following. Output signal was pre-charged to high level (5V) at the beginning of the simulation. \$t\_{sub pHZ}\$ was measured from the control signal 50% point to the output signal 90% point, i.e. 0.5V swing.

Output disable time from low level (\$t\_{sub pLZ}\$) was measured as following. Output signal was set to low level (0V) at the beginning of the simulation. \$t\_{sub pLZ}\$ was measured from the control signal 50% point to the output signal 10% point (0.5V swing).

Enable time to high level (\$t\_{sub pZH}\$) was gained by setting the output signal to low level at the beginning and measured from the control signal 50% point to the output 50% point.

Enable time to low level (\$t\_{sub pZL}\$) was gained by pre-charging the output signal to high level at the beginning and measured from the control signal 50% point to the output 50% point.

For all cells (except pads), fan out load = 1 means loading the cell with one inverter; fan out load = 10 means loading the cell with 10 inverters in parallel.

For the pads, fan out load of 1 means a loading of 5 pF while fan out load of 10 means a loading of 50 pF. During the measurement, output of each pad was load with 2 resistors: one 16.67K resistor from Vdd to output and one 5K resistor from output to Gnd.

Following is a list of the SPICE model parameter values (see SPICE reference manual for parameter description) that were used:

	PMOS	NMOS
LEVEL	2	2
VTO	-0.65	0.65
KP	12E-6	32E-6
GAMMA	0.5	1.1
PHI	0.64	0.68
LAMBDA	0.06	0.05
PB	0.85	0.95
CGSO	4.54E-10	6.06E-10
CGDO	4.54E-10	6.06E-10
CGBO	1.6E-10	1.6E-10
RSH	92.0	15.0
CJ	2.0E-4	3.3E-4
CJSW	4.0E-10	7.0E-10
TOX	7.8E-8	7.8E-8
XJ	1.2E-6	1.6E-6
LD	0.96E-6	1.28E-6
UCRIT	2.8E4	4.0E4
UEXP	0.24	0.14
TPG	-1	1
MJ	0.5	0.5
MJSW	0.33	0.33
NFS	1E11	1E11

## **Appendix C**

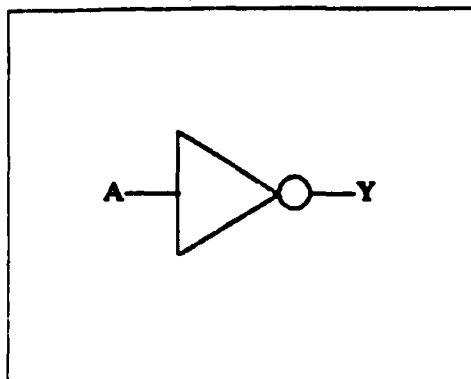
---

### **CMOSPW CELL DESCRIPTIONS**

Herein is contained a description of most of the standard cells available in the CMOSPW cell library. At the end of this appendix is located a writeup on simulation conditions.

# CMOSPW LIBRARY CELL INV

# INVERTER



Truth Table

A	Y
0	1
1	0

$$Y = \bar{A}$$

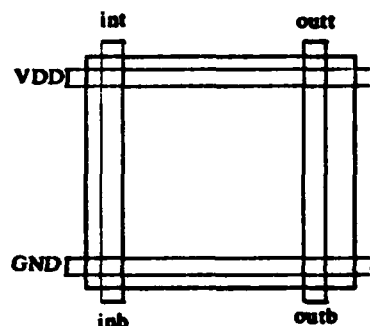
Nearest Functional Equivalent:

CMOS 4049, 4069  
TTL 7404

Nodes	A	Y							
Input Load	1	-							

Block Diagram of I/O pins

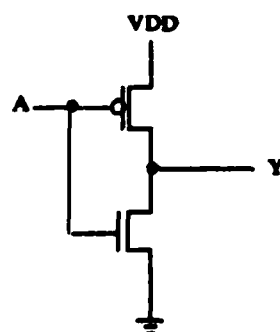
Cell Width: 33  $\lambda$



AC Characteristics  $V_{DD} = 5V$   
Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	$t_{PHL}$	1.5	5.5
Propagation delay (low to high)	$t_{PLH}$	2.0	5.5
Output fall time	$t_{FHL}$	3.0	9.5
Output rise time	$t_{FLH}$	3.0	11.0

Circuit Schematic Diagram



## NOTES:

- 1) rail separation: 58  $\lambda$
- 2) bounding box (x x y): 33 $\lambda$  x 65 $\lambda$

Table values from SPICE simulation

CAESAR file: \$UW\_VLSI\_TOOLS/src/cellib/cmospw/inv.ca

DB files: \$UW\_VLSI\_TOOLS/src/cellib/cmospw/{inv.att,inv.sym}

# CMOSPW LIBRARY CELL NAND2

## 2-INPUT NAND



Truth Table

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

$$Y = \overline{AB}$$

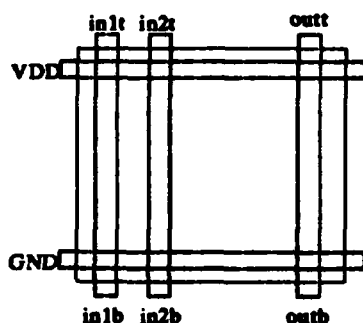
Nearest Functional Equivalent:

CMOS 4011  
TTL 7400

Nodes	A	B	Y						
Input Load	1	1	-						

Block Diagram of I/O pins

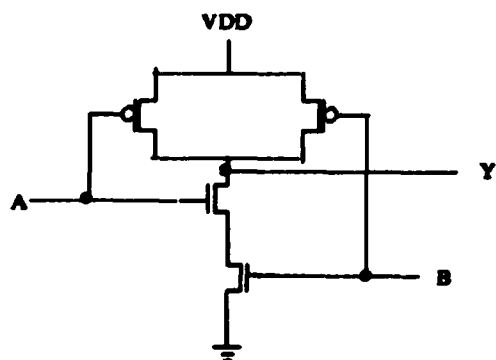
Cell Width: 49  $\lambda$



AC Characteristics  $V_{DD} = 5V$   
Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	$t_{PHL}$	2.5	10.0
Propagation delay (low to high)	$t_{PLH}$	3.0	10.5
Output fall time	$t_{FHL}$	4.0	19.0
Output rise time	$t_{TLH}$	4.5	21.5

Circuit Schematic Diagram



### NOTES:

- 1) rail separation: 58  $\lambda$
- 2) bounding box (x  $\times$  y): 49 $\lambda$   $\times$  65 $\lambda$

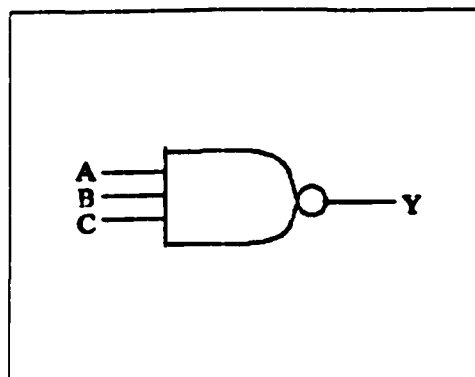
Table values from SPICE simulation

CAESAR file: \$UW\_VLSI\_TOOLS/src/cellib/cmospw/nand2.ca

DB files: \$UW\_VLSI\_TOOLS/src/cellib/cmospw/{nand2.att, nand2.sym}

# CMOSPW LIBRARY CELL NAND3

# 3-INPUT NAND



Truth Table

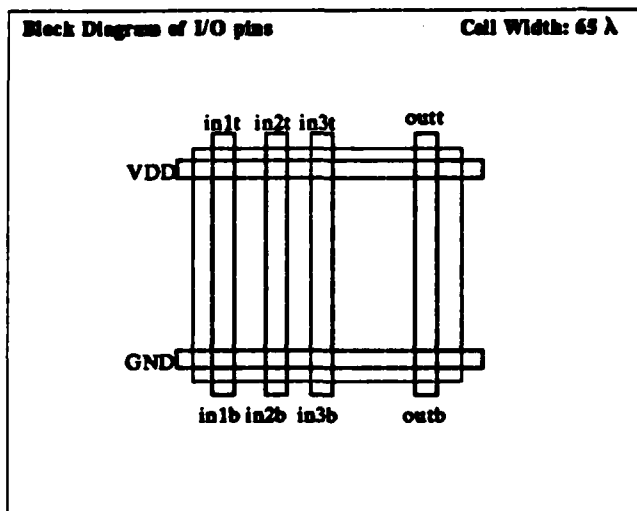
A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

$$Y = \overline{ABC}$$

Nearest Functional Equivalent:

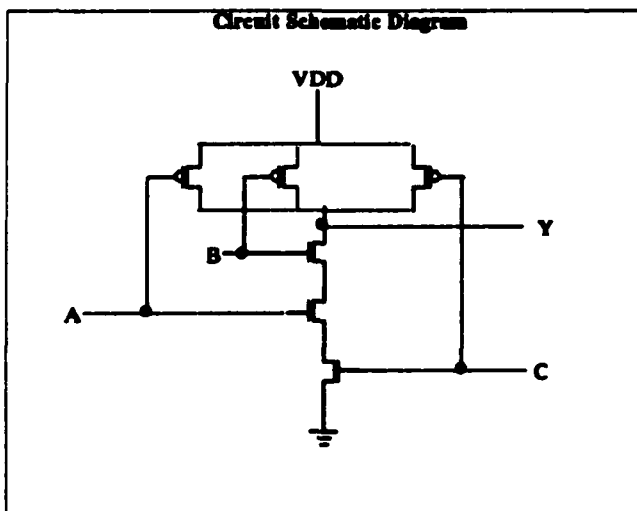
CMOS 4023  
TTL 7410

Nodes	A	B	C	Y					
Input Load	1	1	1	-					



AC Characteristics  $V_{DD} = 5V$   
Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	$t_{PHL}$	2.5	8.5
Propagation delay (low to high)	$t_{PLH}$	3.0	11.5
Output fall time	$t_{THL}$	3.5	17.0
Output rise time	$t_{TLH}$	4.5	21.5



## NOTES:

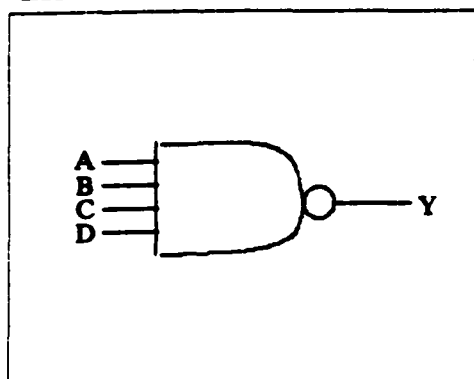
- 1) rail separation: 58  $\lambda$
- 2) bounding box (x  $\times$  y): 65 $\lambda$   $\times$  65 $\lambda$

Table values from SPICE simulation

CAESAR file: \$UW\_VLSI\_TOOLS/src/cellib/cmospw/nand3.ca

DB files: \$UW\_VLSI\_TOOLS/src/cellib/cmospw/{nand3.att, nand3.sym}

# LIBRARY CELL NAND4



## Truth Table

A	B	C	D	Y
0	X	X	X	1
X	0	X	X	1
X	X	0	X	1
X	X	X	0	1
1	1	1	1	0

## 4-INPUT NAND

$$Y = \overline{ABCD}$$

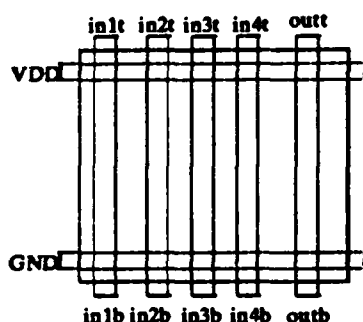
Nearest Functional Equivalent:

CMOS 4012  
TTL 7420

Nodes	A	B	C	D	Y				
Input Load	1	1	1	1	-				

## Block Diagram of I/O pins

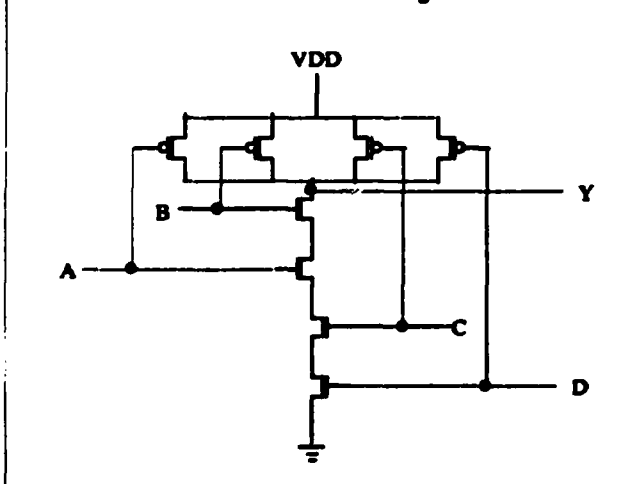
Cell Width: 81  $\lambda$



## AC Characteristics $V_{DD} = 5V$ Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	$t_{PHL}$	3.5	10.5
Propagation delay (low to high)	$t_{PLH}$	3.0	11.0
Output fall time	$t_{FHL}$	5.5	23.0
Output rise time	$t_{TLH}$	5.5	23.5

## Circuit Schematic Diagram



## NOTES:

- 1) rail separation: 58  $\lambda$
- 2) bounding box (x x y): 81 $\lambda$  x 65 $\lambda$

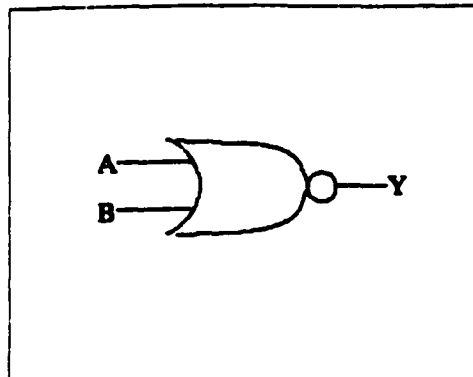
Table values from SPICE simulation

CAESAR file: \$UW\_VLSI\_TOOLS/src/cellib/cmospw/nand4.ca

DB files: \$UW\_VLSI\_TOOLS/src/cellib/cmospw/{nand4.att, nand4.sym}



# LIBRARY CELL NOR2



Truth Table

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

# 2-INPUT NOR

$$Y = \overline{A + B}$$

Nearest Functional Equivalent:

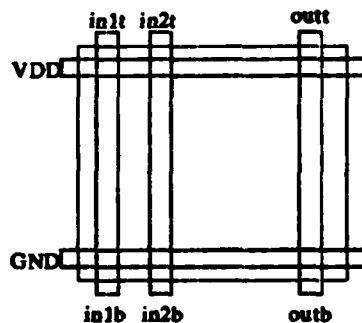
CMOS 4001

TTL 7402

Nodes	A	B	Y						
Input Load	1	1	-						

Block Diagram of I/O pins

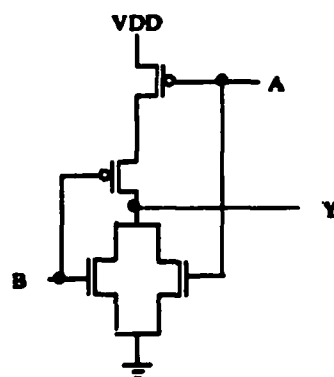
Cell Width: 49  $\lambda$



AC Characteristics  $V_{DD} \sim 5V$   
Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	$t_{PHL}$	2.1	6.0
Propagation delay (low to high)	$t_{PLH}$	2.5	9.0
Output fall time	$t_{THL}$	3.5	10.5
Output rise time	$t_{TLH}$	5.5	19.5

Circuit Schematic Diagram



## NOTES:

- 1) rail separation: 58  $\lambda$
- 2) bounding box (x  $\times$  y): 49 $\lambda$   $\times$  65 $\lambda$

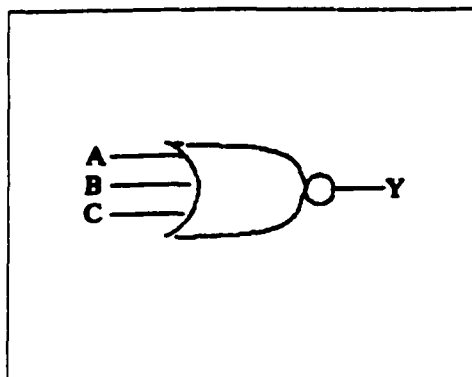
Table values from SPICE simulation

CAESAR file: \$UW\_VLSI\_TOOLS/src/cellib/cmospw/nor2.ca

DB files: \$UW\_VLSI\_TOOLS/src/cellib/cmospw/{nor2.att, nor2.sym}

# LIBRARY CELL NOR3

# 3-INPUT NOR



Truth Table

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

$$Y = \overline{A + B + C}$$

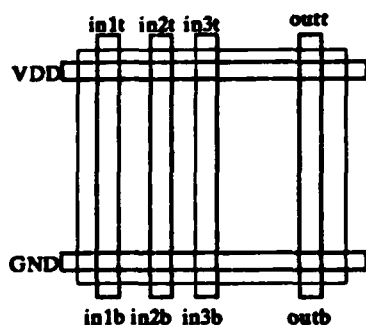
Nearest Functional Equivalent:

CMOS 4000  
TTL 7427

Nodes	A	B	C	Y					
Input Load	1	1	1	-					

Block Diagram of I/O pins

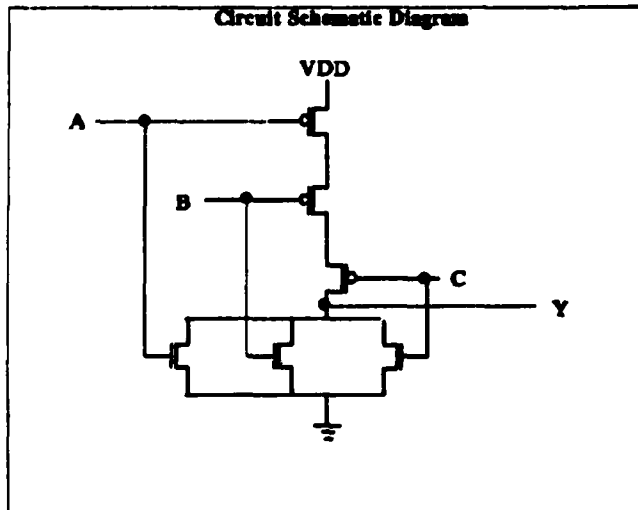
Cell Width: 65  $\lambda$



AC Characteristics  $V_{DD} = 5V$   
Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	$t_{PHL}$	2.0	6.5
Propagation delay (low to high)	$t_{PLH}$	3.1	14.0
Output fall time	$t_{THL}$	3.1	12.0
Output rise time	$t_{TLH}$	6.5	29.0

Circuit Schematic Diagram



## NOTES:

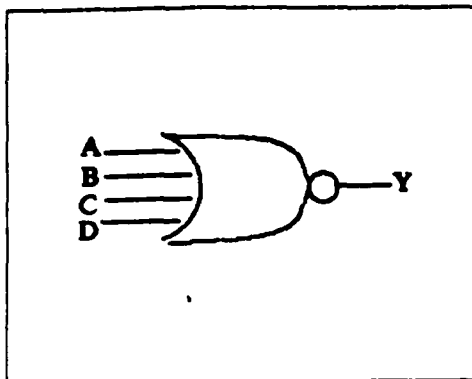
- 1) rail separation: 58  $\lambda$
- 2) bounding box (x  $\times$  y): 65 $\lambda$   $\times$  65 $\lambda$

Table values from SPICE simulation

CAESAR file: \$UW\_VLSI\_TOOLS/src/cellib/cmospw/nor3.ca

DB files: \$UW\_VLSI\_TOOLS/src/cellib/cmospw/{nor3.att, nor3.sym}

# **LIBRARY CELL NOR4**



## **Truth Table**

A	B	C	D	Y
0	0	0	0	1
X	X	X	1	0
X	X	1	X	0
X	1	X	X	0
1	X	X	X	0

## **4-INPUT NOR**

$$Y = \overline{A+B+C+D}$$

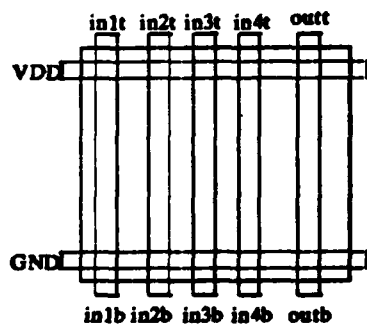
**Nearest Functional Equivalent:**

CMOS 4002  
TTL 7425

Nodes	A	B	C	D	Y				
Input Load	1	1	1	1	-				

## **Block Diagram of I/O pins**

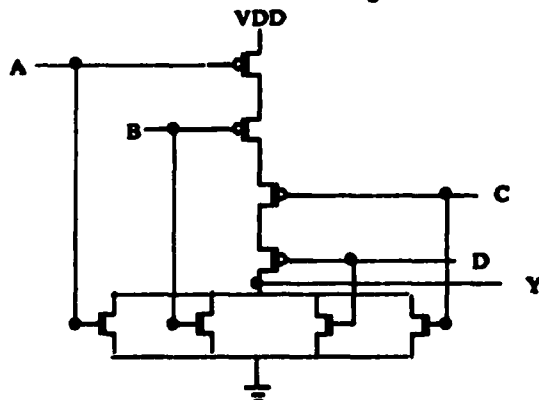
Cell Width: 81  $\lambda$



## **AC Characteristics $V_{DD} = 5V$ Input Transition Time = 5ns**

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	$t_{PHL}$	2.5	6.5
Propagation delay (low to high)	$t_{PLH}$	5.5	19.0
Output fall time	$t_{THL}$	4.0	12.4
Output rise time	$t_{TLH}$	9.5	40.0

## **Circuit Schematic Diagram**



## **NOTES:**

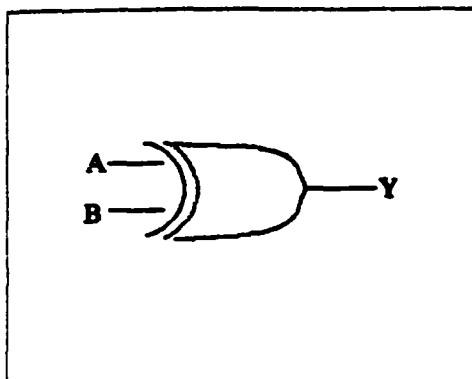
- 1) rail separation: 58  $\lambda$
- 2) bounding box (x  $\times$  y): 81 $\lambda$   $\times$  65 $\lambda$

Table values from SPICE simulation

CAESAR file: \$UW\_VLSI\_TOOLS/src/cellib/cmospw/nor4.ca  
DB files: \$UW\_VLSI\_TOOLS/src/cellib/cmospw/{nor4.att, nor4.sym}

# CMOSPW LIBRARY CELL XOR2

## 2-INPUT XOR



Truth Table

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

$$Y = A \oplus B$$

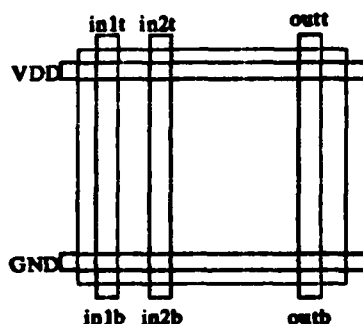
Nearest Functional Equivalent:

CMOS 4030  
TTL 7486

Nodes	A	B	Y						
Input Load	2	2	-						

Block Diagram of I/O pins

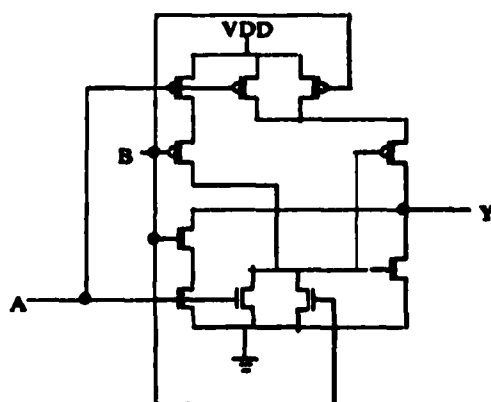
Cell Width: 89  $\lambda$



AC Characteristics  $V_{DD} = 5V$   
Input Transition Time = 5ns

Parameter		Fan Out Load = 1	Fan Out Load = 10
		Typ	Typ
Propagation delay (high to low)	$t_{PHL}$	4.1	7.0
Propagation delay (low to high)	$t_{PLH}$	4.0	13.5
Output fall time	$t_{FHL}$	3.1	7.5
Output rise time	$t_{TLH}$	5.5	27.0

Circuit Schematic Diagram



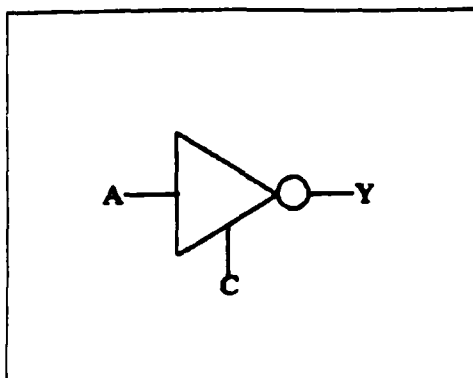
### NOTES:

- 1) rail separation: 58  $\lambda$
- 2) bounding box (x  $\times$  y): 89  $\lambda \times$  65  $\lambda$

Table values from SPICE simulation

CAESAR file: \$UW\_VLSI\_TOOLS/src/cellib/cmospw/xor2.ca  
DB files: \$UW\_VLSI\_TOOLS/src/cellib/cmospw/{xor2.att, xor2.sym}

# LIBRARY CELL CLKINV



# CLOCKED INVERTER

Truth Table

C	C-	A	Y
0	1	0	Z
0	1	1	Z
1	0	0	1
1	0	1	0

$$Y = \bar{A}$$

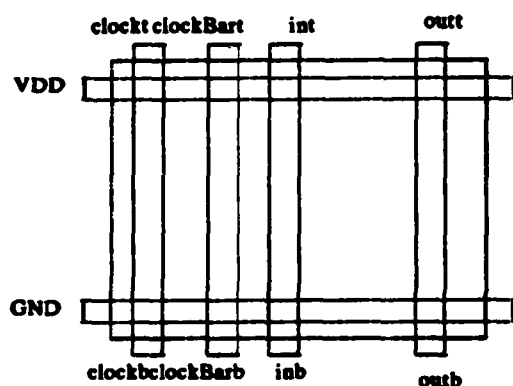
Nearest Functional Equivalent:

CMOS  
TTL

Nodes	C	C-	A	Y						
Input Load	.67	.33	1	-						

Block Diagram of I/O pins

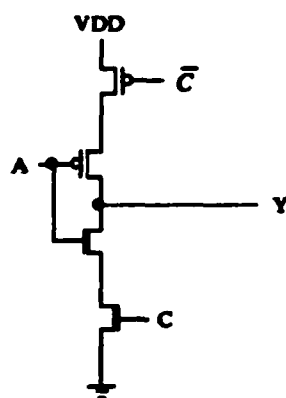
Cell Width: 58  $\lambda$



AC Characteristics  $V_{DD} = 5V$   
Input Transition Time = 5ns

Parameter		Fan Out Load = 1	Fan Out Load = 10
		Typ	Typ
Propagation delay (high to low)	$t_{PHL}$	2.0	7.5
Propagation delay (low to high)	$t_{PLH}$	3.0	12.0
Output fall time	$t_{THL}$	3.0	9.5
Output rise time	$t_{TLH}$	5.0	25.5

Circuit Schematic Diagram



## NOTES:

- 1) rail separation: 58  $\lambda$
- 2) bounding box (x  $\times$  y): 58 $\lambda$   $\times$  65 $\lambda$

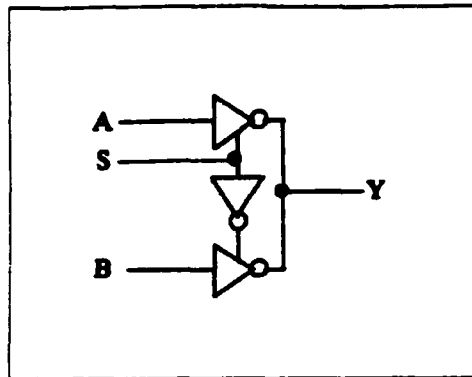
Table values from SPICE simulation

CAESAR file: \$UW\_VLSI\_TOOLS/src/cellib/cmospw/clkinv.ca

DB files: \$UW\_VLSI\_TOOLS/src/cellib/cmospw/{clkinv.att, clkinv.sym}

# LIBRARY CELL SEL2INV

# 2-INPUT INVERTED SELECTOR



Truth Table

S	A	B	Y
1	1	X	0
1	0	X	1
0	X	1	0
0	X	0	1

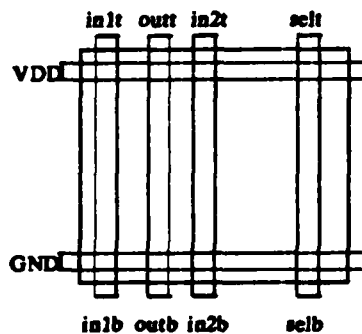
Nearest Functional Equivalent:

CMOS  
TTL 74157

Nodes	S	A	B	Y						
Input Load	2	1	1	-						

Block Diagram of I/O pins

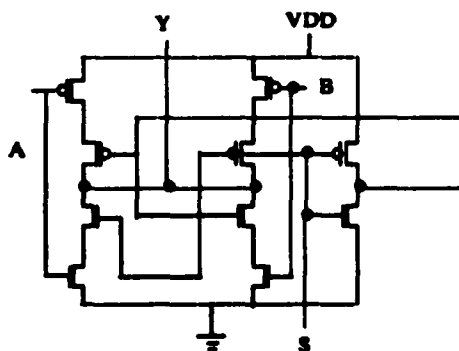
Cell Width: 86  $\lambda$



AC Characteristics  $V_{DD} = 5V$   
Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	$t_{PHL}$	4.5	11.5
Propagation delay (low to high)	$t_{PLH}$	5.1	14.0
Output fall time	$t_{THL}$	5.5	20.5
Output rise time	$t_{TLH}$	6.9	27.5

Circuit Schematic Diagram



## NOTES:

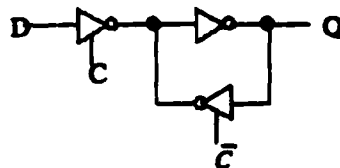
- 1) rail separation: 58  $\lambda$
- 2) bounding box (x x y): 86 $\lambda$  x 65 $\lambda$

Table values from SPICE simulation

CAESAR file: \$UW\_VLSI\_TOOLS/src/cellib/cmospw/sel2inv.ca

DB files: \$UW\_VLSI\_TOOLS/src/cellib/cmospw/{sel2inv.att, sel2inv.sym}

## D-LATCH



### Truth Table

C	C-	D	O
1	0	1	1
1	0	0	0
0	1	X	0

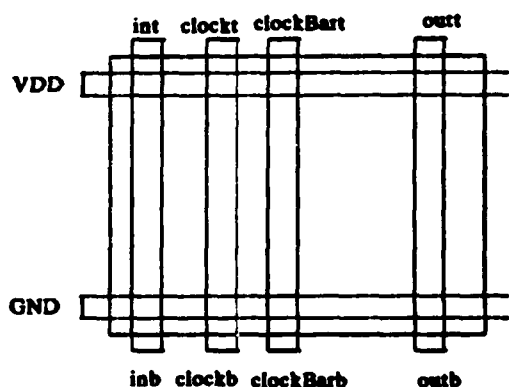
**Nearest Functional Equivalent:**

**CMOS  
TTL 74LS77**

Nodes	C	C-	D	Q						
Input Load	1	1	1	-						

### Block Diagram of I/O pins

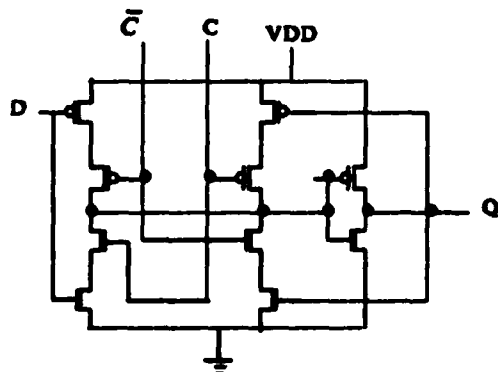
**Cell Width: 100  $\lambda$**



**AC Characteristics  $V_{DD} = 5V$   
Input Transition Time = 5ns**

Parameter		Fan Out Load = 1	Fan Out Load = 10
		Typ	Typ
Propagation delay (high to low)	$t_{PHL}$	5.5	10.0
Propagation delay (low to high)	$t_{PLH}$	5.0	9.5
Output fall time	$t_{FHL}$	4.0	12.0
Output rise time	$t_{TLH}$	4.0	14.5

### Circuit Schematic Diagram



**NOTES:**

- 1) rail separation:  $58 \lambda$   
2) bounding box ( $x \times y$ ):  $100\lambda \times 65\lambda$

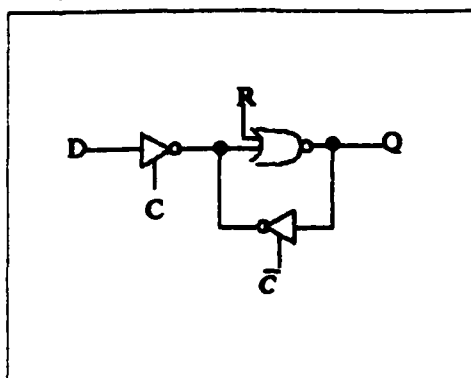
### Table values from SPICE simulation

**CAESAR file:** \$UW\_VLSI\_TOOLS/src/cellib/cmospw/dlatch.ca

**DB files:** `$UW_VLSI_TOOLS/src/cellib/cmospw/{dlatch.att, dlatch.sym}`

# LIBRARY CELL DLATCHR

# D-LATCH WITH RESET



Truth Table

C	C-	R	D	Q
1	0	0	D	D
0	1	0	X	Q
X	X	1	X	0

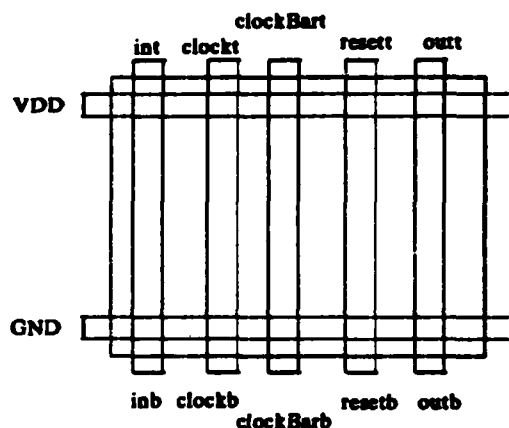
Nearest Functional Equivalent:

CMOS  
TTL

Nodes	C	C-	R	D	Q					
Input Load	1	1	1	1	-					

Block Diagram of I/O pins

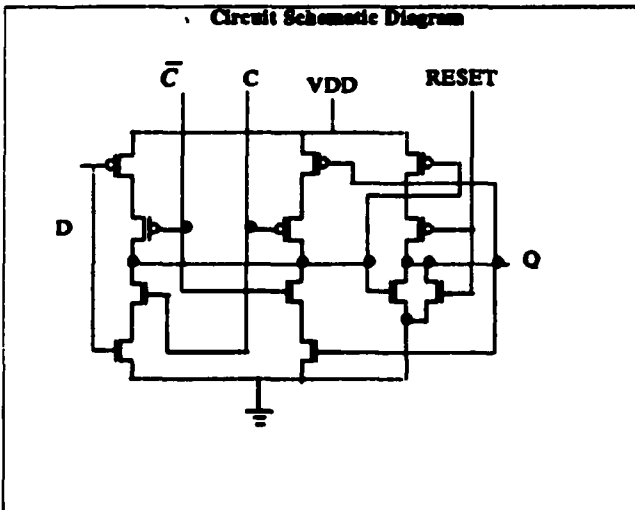
Cell Width: 114  $\lambda$



AC Characteristics  $V_{DD} = 5V$   
Input Transition Time = 5ns

Parameter		Fan Out Load=1	Fan Out Load=10
		Typ	Typ
Propagation delay (high to low)	$t_{PHL}$	5.6	10.0
Propagation delay (low to high)	$t_{PLH}$	6.5	15.6
Output fall time	$t_{THL}$	4.4	12.5
Output rise time	$t_{TLH}$	7.0	24.4

Circuit Schematic Diagram



## NOTES:

- 1) rail separation: 58  $\lambda$
- 2) bounding box (x y): 114 $\lambda$  x 65 $\lambda$
- 3) Reset propagation delay: 2.0 6.0
- 4) Reset fall time: 3.5 11.5

Table values from SPICE simulation

CAESAR file: \$UW\_VLSI\_TOOLS/src/cellib/cmospw/dlatchr.ca

DB files: \$UW\_VLSI\_TOOLS/src/cellib/cmospw/{dlatchr.att, dlatchr.sym}



## Simulation Conditions

All CMOS standard cells were simulated by *spice* (a circuit simulator program). The circuits were generated from the layout artwork by first creating CIF (Caltech Intermediate Form) files from the layout. The circuit extraction program *meatra* was then used to extract the circuit from the CIF file. Following are some electrical data of the timing simulations.

Temperature parameter was set to 27°C

Control signal transition time was 5 ns

VDD = 5V      GND = 0V

Output fall time (\$t\_{sub} THL\$) was measured from 90% (4.5V) point to 10% point (0.5V) of the output signal.

Output rise time (\$t\_{sub} TLH\$) was measured from 10% (0.5V) point to 90% point (4.5V) of the output signal.

Propagation delay (\$t\_{sub} pHL\$ or \$t\_{sub} pLH\$) was measured from the control signal 50% point (2.5V) to the output signal 50% point (2.5V).

Output disable time from high level (\$t\_{sub} pHZ\$) was measured as following. Output signal was pre-charged to high level (5V) at the beginning of the simulation. \$t\_{sub} pHZ\$ was measured from the control signal 50% point to the output signal 90% point, i.e. 0.5V swing.

Output disable time from low level (\$t\_{sub} pLZ\$) was measured as following. Output signal was set to low level (0V) at the beginning of the simulation. \$t\_{sub} pLZ\$ was measured from the control signal 50% point to the output signal 10% point (0.5V swing).

Enable time to high level (\$t\_{sub} pZH\$) was gained by setting the output signal to low level at the beginning and measured from the control signal 50% point to the output 50% point.

Enable time to low level (\$t\_{sub} pZL\$) was gained by pre-charging the output signal to high level at the beginning and measured from the control signal 50% point to the output 50% point.

For all cells (except pads), fan out load = 1 means loading the cell with one inverter; fan out load = 10 means loading the cell with 10 inverters in parallel.

For the pads, fan out load of 1 means a loading of 5 pF while fan out load of 10 means a loading of 50 pF. During the measurement, output of each pad was load with 2 resistors: one 16.67K resistor from Vdd to output and one 5K resistor from output to Gnd.

Following is a list of the SPICE model parameter values (see SPICE reference manual for parameter description) that were used:

	PMOS	NMOS
LEVEL	2	2
VTO	-0.9	0.9
CGSO	4.0E-10	5.2E-10
CGDO	4.0E-10	5.2E-10
CGBO	4.0E-10	5.2E-10
RSH	95.0	20.0
CJ	2.0E-4	3.2E-4
CJSW	4.5E-10	9E-10
JS	1.0E-4	1.0E-4
TOX	5.0E-8	5.0E-8
NSUB	5.0E15	2.5E16
TPG	-1	+1
XJ	6.0E-7	8.0E-7
LD	5.0E-7	6.4E-7
UO	200	450
UCRIT	8.0E4	8.0E4
UEXP	0.15	0.15
UTRA	0.3	0.3
VMAX	5.0E4	5.0E4

# NETLIST User's Guide

UWINW VLSI Consortium

Department of Computer Science  
University of Washington  
Seattle, WA 98195

(This document is based on portions of the document "User's Guide to NET, PRESIM and RNL/NL," by Christopher J. Terman, Laboratory for Computer Science, M.I.T., Cambridge, MA 02139.)

To run NETLIST type

`netlist infile [outfile] [-o] [-s#] [-d#,#] [-e#,#]`

`infile` is the name of the NETLIST input file, if `outfile` is specified, that file is used for output. The options are:

- `-o` old format input. size specifications are taken to be length/width rather than width/length.
- `-tech` Uses `tech` in the technology portion of the units/tech line at the beginning of the simulation file produced (Default is ???, unknown).
- `-units` Sets the number of centi-microns per lambda to `units` (Default is 250).
- `-s#` use specified number as initializer for internal node names; useful when you want to merge the results of separate NETLIST runs.
- `-d#,#` set the default width (first number) and length (second number) for depletion devices. The defaults are 8 and 2.
- `-e#,#` like `-d` except for enhancement devices. The defaults are 2 and 2.
- `-i#,#` like `-d` except for intrinsic devices. The defaults are 2 and 2.
- `-l#,#` like `-d` except for low-power devices. The defaults are 2 and 2.
- `-p#,#` like `-d` except for p-channel devices. The defaults are 2 and 2.

A NETLIST file can insert other NETLIST files by using the include command:

`include filename.net`

The single argument must be a string (i.e., enclosed in quotes).

Available through NETLIST are all the regular built-in functions of RNL (i.e., a subset of standard LISP primitives) -- see RNL documentation for a description of these subroutines. In addition, NETLIST offers some special functions useful for building a description of a transistor net-

work. These functions are described below.

NETLIST is a macro-based language for describing networks of sized transistors. Names in NETLIST refer to nodes, which presumably get inter-connected by the user through transistors. A node name has two forms

*(n width length)*

*n* is the name of the network node, length and width specify a transistor size. This is used in NETLIST constructs where mention of a name causes creation of a transistor.

*n*

*n* is the name of the network node; when transistor sizes are required they are taken from the appropriate defaults

When using a name to refer to a node, it must first be "declared" (this allows typo's to be caught early on). Nodes are declared by using the node statement or the local statement (see below). The node statement looks like:

*(node n1 n2 n3 ...)*

where *n1*, *n2*, etc are the names to be declared. Note: when using structured names (see the repeat statement) only the first component has to be declared.

The interconnect capacitance associated with a node can be specified as follows:

*(capacitance n 1234)*

*(setq pf-sq-micron-of-diffusion 10e-4)*

*(capacitance n (\* 13 pf-sq-micron-of-diffusion))*

The first argument is the name of the node, the second the capacitance in pf (must be a number).

An electrical node can be given several names by using the connect statement:

*(connect n1 n2 n3 ...)*

The names *n1*, *n2*, etc. will all refer to the same electrical node. This statement is useful for connecting i/o signals to the edge of an array generated by a repeat statement.

The voltage threshold for logic high and low states can be set by the NETLIST command threshold:

*(threshold n 0.2 0.8)*

would set the logic low threshold for node *n* to 0.2 (normalized voltage) and the high threshold to 0.8. If no threshold is specified, the node will be given the default thresholds as given in the configuration file for PRESIM (see PRESIM.DOC for details).

The "delay" of a node (the transition times for changes in the node's value) can be specified by user with the delay command:

*(delay n plh phl)*

where *plh* and *phl* are integers specifying the low-to-high transition delay and the high-to-low delay respectively. Delays are specified in RNL time units (1/10th nanosecond). If you do not specify a delay for a node, RNL will calculate one based on the impedance of the driving transistors and the capacitance of the node; user-specified delays override the usual RNL calculation.

*(ratio gate\_ratio)*

set a global parameter *gate\_ratio* for use in *cmand*, *cnor*, *cinvert* and the transistors connected to the input signal in the *clkinv*. Default *gate\_ratio* is 2.0.

Node interconnections are accomplished by one of the following NETLIST statements:

*(trans g s d [w [l]])*  
*(ettrans g s d [w [l]])*

enhancement mode transistor with gate g, source s, and drain d. l and w specify length and width of transistor (can be omitted).

*(dtrans g s d [w [l]])*

like etrans, except depletion mode transistor

*(itrans g s d [w [l]])*

like etrans, except intrinsic transistor

*(ltrans g s d [w [l]])*

like etrans, except low-power transistor

*(ptrans g s d [w [l]])*

like etrans, except p-channel transistor

*(tgate out in node nodebar)*

wires a CMOS transmission gate from the signals node and nodebar. The order is alphabetical, the n type is gated by node and the p type by nodebar. The size of the p type device is set explicitly on the nodes node and nodebar not by the ratio commands. Additional arguments (more control) may be added but there must be an even (2N) number or it will complain. Nodes in and out are have their usual meanings.

*(pullup a)*

depletion-mode pullup (to vdd) of a.

*(pulldown a n-1 ... n-k)*

chain of k transistors from a to gnd, gates of transistor are n-1, ..., n-k.

*(invert a b)*

two-transistor NMOS inverter with output a and input b.

*(cinvert a b)*

two-transistor CMOS inverter with output a and input b. The size of the p type transistor is determined by the current value of ratio. See the command ratio for adjusting this value. Default 2.0.

*(clkinv out in clk clk-)*

CMOS clocked inverter. This function builds a clocked inverter from clk, clk- and in nodes. Clk gates the n type transistor and clk- the p type (just like tgate). The size of the p device gated by in is determined from the current value of gate\_ratio (set by the ratio command). The size of the p device gated by nodes clk and clk- are set using standard node syntax and does not use the ratio command.

*(nor a n-1 ... n-k)*

pulls up a, and creates k transistors from a to gnd controlled by n-1 through n-k.

*(cnor a n-1 ... n-k)*

produces a CMOS nor gate with output a and inputs n-1 ... n-k. The node a is pulled up with a chain p type devices and is connected to gnd with the n type devices. Both sets are gated by the list of inputs. The size of the p type transistor is determined by the current value of ratio. See the command ratio for adjusting this value. Default 2.0.

*(nand a n-1 ... n-k)*

equivalent to

```
(pullup a)
(pulldown a n-1 ... n-k)
```

```
(cnand a n-1 ... n-k)
```

produces a CMOS nand gate with output a and inputs n-1 ... n-k. The node a is connected to Vdd through the p type devices and pulled down by chain of n type devices. Both sets are gated by the list of inputs. The size of the p type transistor is determined by the current value of ratio. See the command ratio for adjusting this value. Default 2.0.

```
(and-or-invert a (n-1 ... n-k) ... (m-1 ... m-l))
```

equivalent to

```
(pullup a)
(pulldown a n-1 ... n-k)
...
(pulldown a m-1 ... m-l)
```

Iteration construct is repeat statement:

```
(repeat index low high
  [(local l-1 ... l-j)]
...)
```

where index will be given successive values starting with low and finishing with high. You can use the index in structured names, e.g.:

```
foo.index foo.(1+ index) foo.(1- index).bar ...
```

local variables are described under macros.

For ease of circuit entry, the user can build and call parameterized macros. macro definitions have the form

```
(macro n (p-1 ... p-k)
  [(local l-1 ... l-j)]
...)
```

where n is the name of the new NETLIST function being created, p-1 ... p-k are the formal parameters, l-1 ... l-j are the optional local node names used in the body.

The macro is invoked as follows:

```
(n a-1 ... a-k)
```

which causes the body to be interpreted after

- 1) all occurrences of p-1 in the body have been replaced by a-1, etc.
- 2) all occurrences of l-1 in the body have been replaced by a new, unique node name. Unique names will be a number (like for anonymous nodes in pulldowns).

### 3.0 Examples

In the following examples

```
e g s d l w
```

specifies an enhancement-mode transistor with gate g, source s, and drain d with length l and width w.

```
d g s d l w
```

is similar, except transistor is depletion mode.

Quickie examples:

```

(invert a b)
d a a vdd 8 2
c b a gnd 2 2

(invert a (b 17 5))
d a a vdd 8 2
c b a gnd 5 17

(invert (a 2 2) (b 2 4))
d a a vdd 2 2
c b a gnd 2 4

(nor (a 16 2) (b 2 4) c d)
d a a vdd 2 16
c b a gnd 4 2
e c a gnd 2 2
e d a gnd 2 2

(and-or-invert a (b c d) (e f) (g))
d a a vdd 8 2
c b a 1001 2 2
e c 1001 1002 2 2
e d 1002 gnd 2 2
e e a 1003 2 2
e f 1003 gnd 2 2
e g a gnd 2 2

```

Two dimensional array of foo's:

```
(repeat i 1 8 (repeat j 1 8 foo.i.j))
```

generates

```

foo.1.1 foo.1.2 foo.1.3 ... foo.1.8
foo.2.1 ... foo.8.8

```

Simple two-inverter dynamic memory cell:

```

(macro bitcell (output output-enb input input-enb refresh)
  (local a b c)
  (trans input-enb input a 2 4)
  (invert b a)
  (invert (c 2 2) (b 2 8))
  (trans refresh a c)
  (trans output-enb c output 2 4)
)

```

```
(bitcell bit0 renb bit0 wenb phi2)
```

generates

```

e wenb bit0 1001 4 2
d 1002 1002 vdd 8 2
e 1001 1002 gnd 2 2
d 1003 1003 vdd 2 2
e 1002 1003 gnd 8 2
e phi2 1001 1003 2 2
e renb 1003 bit0 4 2

```

Assume you had an alu bit-slice macro of the following form

*(alu carry-in operand1 operand2 result carry-out)*

then the following macro would produce an n-bit alu:

```

(macro ALU (n databus1 databus2 resultbus cin cout)
  (connect cin cout 0)
  (repeat i 1 n
    (alu cout.(1-i) databus1.i databus2.i resultbus.i cout.i))
  (connect cout cout.n)
  )

```

Instead of using the connect statement one could have conditionalized the calculation of the arguments to alu:

```

(macro ALU (n databus1 databus2 resultbus cin cout)
  (repeat i 1 n
    (alu (cond ((= i 1) cin) (t cout.(1-i)))
          databus1.i
          databus2.i
          resultbus.i
          (cond ((= i n) cout) (t cout.n))) ))
  )

```

The file /usr/vlsi/nl/pads.net contains the following macros:

```

(input-pad world)           ; the input pad
(output-pad world in)       ; the output pad
(tristate-pad world in direction) ; the tristate pad
(clockbar-pad world ~phi1 ~phi2) ; the clock pad

```



# PRESIM User's Guide

UWINW VLSI Consortium

Department of Computer Science  
University of Washington  
Seattle, WA 98195

(This document is based on portions of the document "User's Guide to NET, PRESIM and RNL/NL," by Christopher J. Terman, Laboratory for Computer Science, M.I.T., Cambridge, MA 02139.)

One must first convert the .sim file to a network file suitable for use by RNL or NL -- to do this we run PRESIM:

```
presim foo.sim foo [config] options...
```

which converts the file foo.sim into a binary file for RNL/NL called foo.

The -g option:

Suppresses the sum-of-products formation. This may be desired if you think sum-of-products is formed wrong otherwise the advantages of the transistor and node reduction make this option unattractive.

The -c option:

*-cfile,minvalue*

writes a list of node names and capacitances to the specified file. Only capacitances larger than minvalue will be included.

The -t option:

*-tfile,minvalue*

writes a list of transistors and RC values to the specified file -- there are two entries for each transistor. The R's come from the size of the transistor, C's from the source/drain capacitance. Only RC values larger than minvalue will be included.

The -p option:

*-presist,voltage*

provides a worse-case estimate of the circuit power consumption by assuming that all the pullups (DEP or LOWP devices with drain=VDD) are all on simultaneously. "Voltage" specifies the supply

voltage, for example "-p5" specifies a VDD of 5 volts. The result is printed after PRESIM completes its other processing. When figuring the resistance of a pullup device the "power" characteristic resistance as set in the config file is used.

The optional third file (config) specifies various electrical parameters. The internal values (the defaults) are a generic set. They do not reflect any particular fabrication process. (UW-NW VLSI NOTE: A configuration file is provided in the source code that duplicates the internal settings as an example of how this file could be used. In addition we note that, the resistor values are stored first sorted by width, then by length not by the ratio. Values not explicitly provided in the configuration file are estimated by linear interpolation.) The format of this file is lines of the form

*parameter value comments...*

Lines beginning with ";" are treated as all comment. The parameter names and their default values are:

; configuration file for "standard" MPC process

```
capm2a .00000 ; 2nd metal capacitance -- area, pf/sq-micron
capm2p .00000 ; 2nd metal capacitance -- perimeter, pf/micron
capma .00003 ; 1st metal capacitance -- area, pf/sq-micron
capmp .00000 ; 1st metal capacitance -- perimeter, pf/micron
cappa .00004 ; poly capacitance -- area, pf/sq-micron
cappp .00000 ; poly capacitance -- perimeter, pf/micron
capda .00010 ; n-diffusion capacitance -- area, pf/sq-micron
capdp .00050 ; n-diffusion capacitance -- perimeter, pf/micron
cappda .00010 ; p-diffusion capacitance -- area, pf/sq-micron
cappdp .00060 ; p-diffusion capacitance -- perimeter, pf/micron
capga .00040 ; gate capacitance -- area, pf/sq-micron
```

```
lambda 2.5 ; microns/lambda (conversion from .sim file units
; to units used in cap parameters)
```

```
lowthresh 0.3 ; logic low threshold as a normalized voltage
highthresh 0.8 ; logic high threshold as a normalized voltage
```

```
cntpullup 0 ; < > 0 means that the capacitor formed by gate of
; pullup should be included in capacitance of output
; node
```

```
diffperim 0 ; < > 0 means do not include diffusion perimeters
; that border on transistor gates when figuring
; sidewall capacitance (*)
```

```
subparea 0 ; < > 0 means that poly over transistor region will not
; be counted as part of the poly-bulk capacitor (*)
```

**diffext 0** ; diffusion extension for each transistor, i.e., each  
 ; transistor is assumed to have a rectangular source  
 ; and drain diffusion extending diffext units wide and  
 ; transistor-width units high. The effect of the  
 ; diffusion extension is to add some capacitance to  
 ; the source and drain node of each transistor --  
 ; useful when processing the output of NET to improve  
 ; the capacitive loading approximations without adding  
 ; explicit load capacitors. diffext is specified in  
 ; lambda (it will be converted using the lambda factor  
 ; above).

**resistance channel context width length resist**  
 ; this command specifies the equivalent resistance for a transistor  
 ; of type channel with the specified width and length. Transistors  
 ; matching this entry will have the specified resistance; linear  
 ; interpolation is done if the width and/or length is not matched  
 ; exactly.  
 ; channel is one of "enh", "dep", "intrinsic", "low-power",  
 ; "pullup", or "p-chan"  
 ; context is one of "static", "dynamic-high", "dynamic-low", or "power"  
 ; width is given in lambda  
 ; length is given in lambda  
 ; resist is given in ohms

(\*) These parameters should be 1 only when processing the output of the node extractor. They cause various corrections to be made to the interconnect component of a node's capacitance -- usually only extracted .sim files have information regarding interconnect capacitance.

PRESIM uses these parameters in calculating the capacitance for each electrical node and the resistance for each transistor channel.

# RNL 4.2 User's Guide

UW/NW VLSI Consortium  
Sieg Hall, FR-35,  
University of Washington,  
Seattle, WA 98195

(This manual documents version 4.2 (UW) RNL. The manual is based on  
Chris Terman's manual of similar title.)

## 1. INTRODUCTION

RNL is a timing logic simulator for digital MOS circuits. It is an event driven simulator that uses a simple RC (resistance capacitance) model of the circuit to estimate node transition times and to estimate the effects of charge sharing. The user interface is a simple LISP interpreter. This allows both interactive simulation and the programming of complex simulations. See Chapter 2 of "Simulation Tools for LSI Design" by C. Terman for details of the algorithm. A short introduction to the model is included in the "Theory of Operation" section of this guide.

The version of RNL described herein is version 4.2 as distributed by the UW/NW VLSI Consortium. It differs from previous versions in that it is considerably faster for many simulations. In addition the user interface has been augmented.

To use RNL, one needs .sim file for the circuit to be simulated. This can be extracted from the mask file (e.g., CIF) or developed using NETLIST, a program that processes textual schematics.

The first step is to convert the .sim file to a network file suitable for use by RNL by running PRESIM:

```
% presim foo.sim foo [confile] [-cfile,min] [-tfile,min] [presist,voltage] < CR>
```

Presim converts the file "foo.sim" into a binary file for RNL called "foo". The other parameters are optional and are described in detail elsewhere. The conversion process involves the computation of the effective resistances of the transistors as well as the capacitances of the circuit nodes. In order to have a consistent estimation of capacitances we recommend that if you are using the circuit extractor mextra that you use the "-o" option to force the program to output the dimensions of the circuit nodes rather than to estimate their capacitances. (See the PRESIM documents for information on options and sections of this manual on calibration.)

To invoke RNL, either type

```
% rnl < CR>
```

or

**% rnl cmdfile <CR>**

If the "cmdfile" argument is provided then it should be the name of a file that contains a sequence of RNL commands. At the very least this command file should load one or more libraries of standard functions and should read in the binary description of the circuit prepared by PRESIM. For all but the simplest of circuits the command file will also contain commands and the definitions of LISP functions written by you to help in your simulation by performing such tasks as test vector generation and the simulation of the environment in which your circuit is designed to operate.

A minimal command file would contain the commands:

```
(load "uwstd.l")
(load "uwsim.l")
(read-network "foo")
```

where the file "foo" was prepared from a ".sim" file by PRESIM.

When the end of the command file is reached, input is taken from the standard input.

## 2. RNL THEORY OF OPERATION

It is not necessary to have a detailed understanding of the internal computations of RNL in order to begin to use it. It is, however, useful to have a general idea of what is going on. In particular, this section will be helpful when reading the discussion on some of the limitations of RNL's circuit model. The rest of this section is a discussion of RNL's internal computations and is quoted directly from C. Terman's original RNL User's Guide.

The RNL simulator are designed to handle ratioed logic, bidirectionality, and charge sharing/storage. They can be used to determine the functionality and approximate timing behavior of circuits commonly found in digital MOS designs.

RNL uses the following simple recipe for simulating a circuit. Recall that PRESIM has established the capacitance of each node and the size of each transistor. (The network extractor written by C. Baker automatically derives both from the mask files; if the network is derived from a NETLIST description, the user must explicitly specify the interconnect capacitance for nodes where it is important.)

Once input values have been assigned by the user, RNL calculates the effects of the new values by repeating the following operations until no further nodes change values:

- (1) when nodes are added to the network (the result of some transistor turning on), compute the "charge sharing" implications of the new node's capacitance and logic state on its electrical neighbors.
- (2) for each node that might be affected calculate  $V_{thv}$  and  $R_{thv}$ , the parameters for the Thevenin equivalent circuit. The new logic state of the node is determined from  $V_{thv}$ .
- (3) if the node has changed state, calculate the transition time using the node's capacitance.
- (4) propagate changes (if any) to other nodes.

Basic to the operation of the simulators is the notion of an event -- an event specifies (i) a node in the network, (ii) a new logic state, and (iii) a time at which the node's value is changed to the new logic state. RNL maintains a list of events, sorted by time, that tells what processing remains to be done. Whenever the user changes an input, an event is added to the list; when the list is empty the network has "settled" and RNL waits for further input.

When started with an initial list, RNL sequentially processes the next event on the list, stopping (1) when the list is empty, (2) when a node the user is tracing changes value, or (3) when the specified amount of simulated time has elapsed. Processing an event entails

- (a) removing the event from the event list.
- (b) changing the node's state to reflect its new value.
- (c) calculating any consequences, i.e., new events, resulting from the node's new value. First all nodes that might be affected by the change are found and marked -- this includes the source and drain nodes of transistors with the current node as a gate, and all nodes connected to these nodes by conducting transistors (the search through the network stops only when an input or a non-conducting transistor is reached). For each marked node two calculations are made: first a "charge sharing" calculation is performed (see 2.1) to model changes of state due to charging/discharging of the node capacitances. Second, a "final value" calculation is done (see section 2.2) to determine the nodes ultimate logical state.

Since nodes are only added to the event list when their values change, portions of the circuit unaffected by the current set of changes to the inputs are not re-evaluated -- the algorithm is event-driven (sometimes called selective trace).

A node can have up to two events pending:

- (1) a "charge sharing" event describing an immediate change in the node's state due to the redistribution of charge among the capacitors for nodes on the connection list. This type of event is only generated when a node is added to a subnetwork (i.e., when a transistor turns on).
- (2) a "final value" event describing what the final, driven state of the node will be.

The simulation computation computes both types of events for each node and then does the following:

- (a) when a new charge sharing event is scheduled, throw away pending events of either flavor. If the new charge sharing event is for the same value that the node currently has, it can be thrown away too, i.e., the node will end up with no events pending.
- (b) when a new final value event is scheduled it will be ignored if
  - (i) there is a pending final value event for the same value which is scheduled to happen at an earlier time than the new event. If this test fails, any pending final event is discarded, and the remaining conditions checked.
  - (ii) there is a pending charge sharing event for the same value as the new final value event.
  - (iii) there is no charge sharing event and the new event is for the same value that the node currently has.

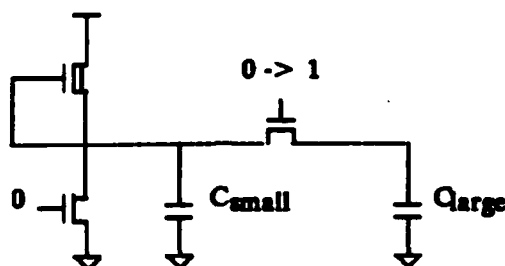
If none of the tests are successful, the new final value event is added to the event list.

These rules are based on the observation that the event that was last calculated reflects the latest network configuration and hence should override events calculated earlier. Charge sharing events throw away final value events since the charge sharing calculation is immediately followed by a new final value calculation.

The next two sections describe the two parts of the simulation computation.

## 2.1. Charge sharing computation

This portion of the simulation calculation tries to model various capacitive effects that happen when two (or more) previously unconnected nodes become connected. For example:



In this circuit the transfer gate has just turned on, connecting a bus (represented by  $C_{large}$ , initially at logic low) with an inverter whose output is a logic high. If  $C_{large}$  and the pass gate are large enough, the inverter output will go low ( $C_{small}$  is discharged) initially, but eventually both the inverter output and bus will go to a logic high. In RNL, this sequence of events happens in two steps: a charge sharing calculation that predicts the first transition, and a final value calculation that predicts the ultimate state.

The charge sharing computation proceeds as follows:

- (1) set  $C_L = C_X = C_H = 0$ ;
- (2) compute connection list: starting with current node, include all nodes in the network that can be reached via non-off transistors (includes transistors with gates with logic state "X"). For the charge sharing calculation, depletion transistors are considered to be "off" since they (usually) represent a high impedance connection over which charge sharing would happen very slowly.
- (3) visiting each node on connection list, calculate summary capacitances ( $C_L$ ,  $C_X$ ,  $C_H$ : each node contributes to the sum corresponding to the node's current state). Actually steps (2) and (3) can be merged into a single computation.
- (4) compute initial state:

$$\text{INITIAL STATE} = \begin{cases} 1 & C_H / (C_L + C_X + C_H) > V_{high} \\ 0 & (C_H + C_X) / (C_L + C_X + C_H) < V_{low} \\ X & \text{otherwise} \end{cases}$$

For each node on the connection list, schedule a transition to the initial state with zero delay – this event may be ignored under the conditions described at the end of the previous section.

$V_{high}$  is the logic high threshold of the node,  $V_{low}$  the logic low threshold; these can be set separately for each node or one can use the default setting (see NETLIST and PRESIM documentation).

Note that although the computation could be made node-by-node, groups of electrically connected nodes are dealt with as a whole since their events are obviously related.

## 2.2. Final value computation

After the charge sharing calculation is done, RNL revisits each group of affected nodes to compute their final values. As we saw in the example of the previous section, a node's ultimate value may differ from its charge-sharing value.

The final value computation computes two pieces of information about each node.

- (1) its final logic state. Recall that the transistor network containing the node is being modeled by an equivalent resistor network. To determine the logic state of a node, RNL computes the Thevenin equivalent for the node in question from the modeling resistor network (more on how this is done below) – the Thevenin equivalent voltage is used to calculate the final logic state of the node.

- (2) if the node value is changing, an estimate of the transition time is needed. If the transition is from high to low, RNL computes the effective resistance to GND for the node ( $R_{GND}$ ) and then calculates the transition time as  $R_{GND} \cdot C$  (capacitance not already at GND). A similar calculation is made for low to high transitions. Transitions to X are defined to take the same time as the shorter of the high-to-low and low-to-high transitions.

The following subsections deal with each part of the final value computation.

### 2.3. Network analysis

This section outlines how the Thevenin equivalent circuit for a given node is calculated from a larger network. We start by describing the simple transistor model, show how the Thevenin equivalent is derived using information about each transistor, and end by showing how the new value of a node is computed.

The transistor model in RNL can be quite simple since it is only used to predict the final logic state of a node and how long each state transition takes. Although the channel resistances of the transistors change as their terminal voltages vary, it might be possible to use "average channel resistances" to characterize the transistors' behavior. RNL does just this -- transistors are modeled as resistors whose resistances are determined by the logic state of the transistor's terminal nodes and the type of transistor:


$R_{transistor} = (\text{length}/\text{width}) \cdot \text{type} \cdot \text{state}$  where

width, length are the dimensions of the active transistor area.

type is the average channel resistance per unit area for the particular type of transistor.

state a scale factor that depends on the logic state of the transistor's terminal nodes.

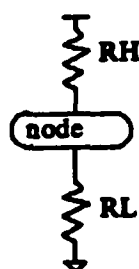
The following table shows type\*state for an enhancement transistor ( $V_g$  is the logic state of the gate node).

	$V_g$	type * state
enhancement 	0	$\infty$
	1	enh
	X	[enh, $\infty$ ]

where enh is the characteristic channel resistance of an enhancement device. When the state of the gate and/or source nodes of a transistor are X, the resistance of the transistor is also "unknown" and is specified by an interval.

We can now describe how  $V_{thrv}$  and  $R_{thrv}$  for a given node can be calculated from a network of nodes and transistors. The network analysis subroutine does a tree walk of the network returning the values of the two resistors,  $R_H$  and  $R_L$ , that make up the characteristic voltage divider for a node:





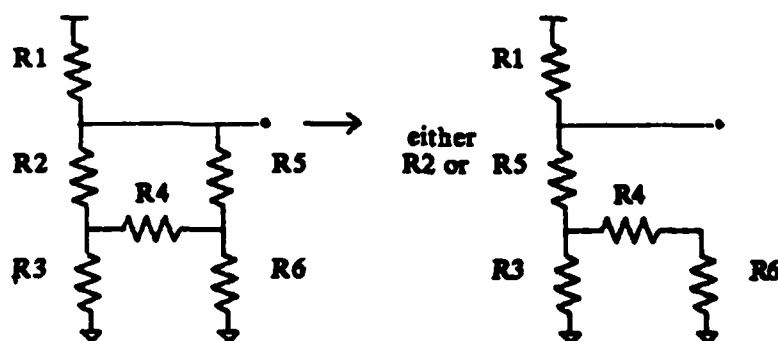
The subroutine is outlined below. The terms "source" and "drain" are used to distinguish between the two terminal nodes of a transistor and do not imply anything about their relative potential.

```

if node is a logic low input {
  return with  $R_H = \infty$  and  $R_L = 0+$ 
}else if node is a logic high input{
  return with  $R_H = 0+$  and  $R_L = \infty$ 
}else{
  local_RH: = local_RL : =  $\infty$ 
  mark current node
  for each "on" transistor L with source connection to current node{
    if drain node is not marked{
      recursively analyze drain node
      derive a voltage divider that approximates the effect of the
      drain node on the current node (approximation uses  $R_H$  and  $R_L$ 
      for the drain node and the equivalent resistance for t)
      parallel approximating voltage divider with local_RH and local_RL
    }
  }
  return with  $R_H = \text{local\_RH}$  and  $R_L = \text{local\_RL}$  }

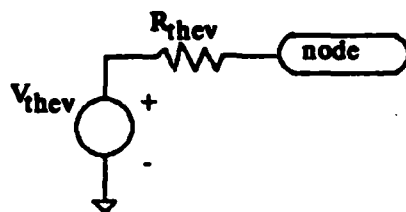
```

Cycles are avoided by marking each node as it is visited: this keeps the tree walk expanding outward from the starting node. If the network does contain cycles, the subroutine only approximates the true resistance to VDD and GND. For example, consider the following gate where the output (the pulled-up node) is the node of interest:



In the circuit on the left the pulldown path contains a cycle; RNL treats the cycle as if the circuit looked as shown on the right. This approximation avoids having to solve a system of equations at simulation time; fortunately, very few networks actually contain such cycles. It is also worth remembering that the resistor network is itself only an approximation -- it is not worth a large investment of computation time to calculate an exact equivalent to the resistor network.

The final state of a node can be characterized by a voltage source with a series resistor, i.e., the Thevenin circuit equivalent for all pieces of the network that influence the value of the given node.



$V_{Thev}$  a voltage interval  $[V_-, V_+]$  in the range  $[0,1]$  specifying the possible voltages the output node may have.

$R_{Thev}$  a resistance interval  $[R_-, R_+]$  in the range  $[0+, \infty]$ .

$V_{Thev}$  and  $R_{Thev}$  are, in general, intervals since the equivalent transistor resistances from which they are derived might themselves lie in an interval. Using the values returned by the network analysis subroutine, we have:

$$V_- = \frac{RL_-}{RL_- + RH_+} \quad \text{and} \quad V_+ = \frac{RL_+}{RL_+ + RH_-}$$

$$R_{Thev} = RL_+ || RH_+$$

Because we are interested only in the worst case determination of the voltage level, we need only consider the worst case Thevenin resistance. As will be seen in the next section, the final value is directly related to  $V_-$  and  $V_+$ .

Different values for  $R_{Thev}$  lead to different conclusions about the state of the node:

*input* ( $R_{Thev} = 0+$ ). Node is designated input node (e.g., VDD or GND). The value of input nodes can only be changed by explicit simulator commands -- the assumption is that they supply enough current to be unaffected by connection (even shorts to other inputs) made by transistors. *driven* ( $0+ < R_{Thev} < \infty$ ). Node is part of a voltage divider between two inputs, i.e., it is connected by transistors to other driven or input nodes. As will be seen below, the logic state of a driven node is determined by  $V_{Thev}$ . *charged* ( $R_{Thev} = \infty$ ). Node is connected, if at all, only to other charged nodes. Charged nodes will maintain their current logic state until either (1) reconnected to some other part of the network, or (2) a user-specified decay interval elapses at which time logic state changes to X.

#### 2.4. Calculating transition delays

Once the final logic state of a node has been determined using the Thevenin equivalents, RNL calculates transition times using one of the following characteristic resistances calculated for each node:

$R_{GND}$  the effective resistance of all direct paths to GND. A simple serial/parallel calculation is used to determine  $R_{GND}$ .

$R_{VDD}$  the effective resistance to  $R_{VDD}$ , computed in a similar fashion.

These two resistances can be calculated at the same time as the Thevenin equivalents; the network analysis routines actually return four values: the intervals  $RL$  and  $RH$ , and the values  $R_{GND}$  and  $R_{VDD}$ . The calculation proceeds as follows:

- (1) set *inputseen* = false,  $C_L = C_X = C_H = 0$ ;
- (2) calculate connection list and summary capacitances ( $C_L, C_X, C_H$ : each node contributes to the sum corresponding to the node's current state). If an output node is reached during the construction of the connection list, set *inputseen* = true.
- (3) if *inputseen* is false, schedule a decay transition for each node on the connection list, then exit.
- (4) if *inputseen* is true, for each node on the connection list:

- (a) if node is an input, continue with next node on list.
- (b) calculate  $V_{thr}$  (also  $R_{GND}$  and  $R_{VDD}$  to be used later).
- (c) compute the node's final state:

$$\text{FINAL STATE} = \begin{cases} 1 & V > V_{Mph} \\ 0 & V < V_{low} \\ X & \text{otherwise} \end{cases}$$

and the effective capacitance and resistance

$$C_{eff} = \begin{cases} C_L + C_X & \text{if final state} = 1 \\ C_H + C_X & \text{if final state} = 0 \\ \min(C_L + C_X, C_H + C_X) & \text{otherwise} \end{cases}$$

$$R_{eff} = \begin{cases} R_{VDD} & \text{if final state} = 1 \\ R_{GND} & \text{if final state} = 0 \\ \min(R_{GND}, R_{VDD}) & \text{otherwise} \end{cases}$$

- (d) schedule a transition to the final state with a delay of  $R_{eff} * C_{eff}$  nanoseconds, or use user-specified delay if present.

Note that the effective capacitance,  $C_{eff}$ , depends on the summary capacitances, not just the capacitance of the node in question. This means that none of the connected nodes will reach its final value much before the others.

## 2.5. Calibrating the model

The charge sharing calculation described in section 2.1 depends only on the capacitance associated with each node. These capacitances are specified by the designer as part of the NETLIST description or in the PRESIM parameter file, both of which are described elsewhere in this document.

The final value computation uses both the node capacitances and resistance information about each transistor. The circuit data base contains the size and type of each transistor -- what the designer must provide in addition is the characteristic resistance for each type of channel (i.e., the resistance of a square transistor of that type). See the description of the set-params subr in section 7.5 for how this information is specified to RNL.

Actually, RNL uses three characteristic resistances:

a *static resistance* used in calculating RH and RL. a *dynamic-low resistance* used in calculating the resistance of paths to GND. a *dynamic-high resistance* used in calculating the resistance of paths to VDD.

A single characteristic resistance won't suffice: RNL uses resistances to determine both the voltage level and transition times -- a resistance value that gives an accurate estimate of the voltage level may not necessarily result in good transition time estimates. Thus, "static" resistances used for voltage level calculations can be specified separately from "dynamic" resistances used for transition time calculations.

There are two sets of dynamic resistances: dynamic-high resistances used when calculating the resistance of paths to VDD, and dynamic-low resistances used when calculating resistance to GND. Ordinarily, these are set to the same value for a particular type of transistor; some useful exceptions:

- (1) setting the dynamic-low resistances very high for devices which should not appear in pulldown paths. The very high transition times that result will serve to flag "strange" circuits.
- (2) setting the dynamic-low resistance for enhancement devices to be appropriate for pulldowns, while setting their dynamic-high resistance to be correct for source-follower configurations.
- (3) since pullups are treated as separate types, the dynamic-high resistance for depletion devices can be set for a source-follower configuration.

Future versions of the RNL will distinguish between different circuit contexts for the same type of device: e.g., enhancement devices would be classed as ordinary, pulldown, source-follower, transfer, etc. Having the ability to set separate dynamic resistances for a transfer device (for example) means that the transition times for high-going and low-going transitions involving the transfer device can be much more accurate.

The static resistances can be estimated from measurements (actual or SPICE'd) of the low threshold for standard logic gates -- there is considerable flexibility since there are many more adjustable parameters than are needed. Dynamic resistances can be estimated by measuring high- and low-going transition times of standard circuit configurations and choosing the characteristic resistances to give an R-C time constant equal to the time the actual waveform takes to cross the desired threshold.

### 3. RNL CALIBRATION VIA THE PRESIM CONFIG FILE

Provided here is a brief description for setting the parameters in the presim configuration file. This is not the only way to obtain these values but the scheme does provide some consistency between analysis models like those used in SPICE. Throughout it is assumed that the Presim User's Guide has been consulted and is available for parameter names, defaults etc.

#### 3.1. Capacitance

There are three basic types of capacitance values that can be set by the use of the configuration file.

- 1) Capacitance from the area of the node interconnect. This case breaks down into 3 subcases; metal area (1st and 2nd layers), polysilicon area and diffusion area (both types in CMOS).
- 2) Capacitance from the perimeter of the node interconnect. Parameters for all layers are provided by presim.
- 3) Capacitance from the area of the gate regions of a node.

All capacitance is assumed grounded.

##### 3.1.1. Area Capacitance

In NMOS the diffusion area capacitance can be estimated as directly proportional to the SPICE model parameter  $C_j$  with proportionality constant  $K_{EQ}$ . For abrupt junctions (a good approximation considering)  $K_{EQ}$  is given by,

$$K_{EQ} = 2 \frac{\phi^{1/2}}{V_2 - V_1} (\sqrt{\phi + V_2} - \sqrt{\phi + V_1}).$$

$V_1 - V_2$  is the voltage range and can be assumed rail to rail. One must also be careful that the units are correct for presim (presim: pF/micron, SPICE: F/m). For a complete discussion of this approximation see "Analysis and Design of Digital Integrated Circuits," D. A. Hodges and H. G. Jackson, McGraw-Hill Book Co., New York, p. 137.

Similarly in CMOS, one uses the  $C_j$  for the two types of diffusion  $n^+$  and  $p^+$ . The contribution

of metal and polysilicon areas can be assumed to be an order of magnitude smaller than diffusion. As of yet we have no experience with second layers of poly and metal.

### 3.1.2. Perimeter Capacitance

For the diffusion perimeter contributions one uses the values for CJSW provided in the SPICE models. Warning, get the units (presim: pF/micron, SPICE: F/m) correct!

### 3.1.3. Gate Capacitance

Gate capacitance can be estimated from ratio of the silicon permittivity and the oxide thickness times the area of the active gate,

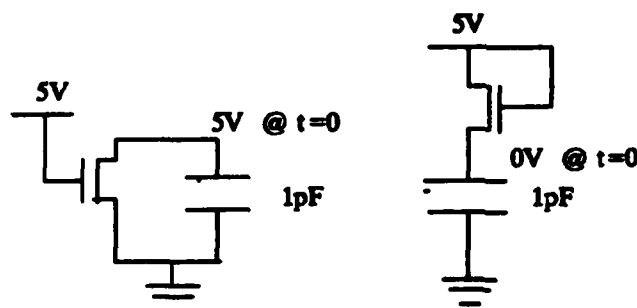
$$C_{gate} = \frac{\epsilon_{Si}}{t_{ox}} LW$$

Again one must be aware to the difference in the units used in presim and SPICE.

## 3.2. Resistance values

Establishing the resistance values is much more complicated. As the circuit elements (e.g. static logic, plas etc.) have a direct bearing on the representation of transistors by the resistors  $R_{dynlow}$  and  $R_{dynhigh}$ .

In most cases it is sufficient to perform circuit analysis on single transistors charging a fixed capacitive load. Two examples that should be included in any suite are shown below.



The type of transistor used in these experiments would vary from depletion, p type enhancements or n type enhancements. Of course the gate voltages and must be adjusted for these various transistor types.

The resistance value is defined then as,

$$R = \frac{\delta t}{C_{load}}$$

This is in effect inverting the calculation done by RNL. The R values can be computed for the all types and the typical sizes (length and width) of transistors used in the circuits to be simulated by RNL. This should be stressed, the table maintained by presim is indexed by type and the length and width independently not by the ratio  $\frac{L}{W}$ .

Interestingly as a practical matter using the above definition for the resistance includes some of the effect of the voltage dependent capacitance. This can be represented by the writing  $\delta t$  as

$$\delta t = R(C_{load} + C(V)_{parasitic}).$$

From this the ratio  $\frac{C(V)_{parasitic}}{C_{load}}$  is the contribution to the resistance R from the voltage dependent

capacitance.

#### 4. SOME OBSERVATIONS ON THE LIMITATIONS OF RNL'S CIRCUIT MODEL

We begin this section quoting from the original User's Guide by C. Terman.

*It should be remembered that the programs are based on a model of what actually happens. As with any model, there are likely to be discrepancies between the predictions of the model and what actually happens. The tools described here try hard to be conservative, i.e. give a pessimistic prediction -- but this can't be guaranteed. Thus, it's wise to acquaint oneself with how the models work and where their shortcomings lie; think of the tools as performing a calculation you could do by hand (only a lot faster and with greater accuracy and consistency); for your own protection, don't treat the tools as black boxes.*

With this warning in mind it will be assumed that the reader has acquainted themselves with the model. The basics are provided in the Theory of Operation section of this user's guide.

As a practical matter RNL provides sufficient parameters for nodes and circuit elements to reproduce the overall behavior obtained from other more elaborate circuit analysis tools. However, it should be remembered that as the designer pushes the tolerances no simulation may reflect the physical device.

##### 4.1. Propagation of X States

The main considerations for X state evaluation are;

- 1) Initial resistance values ( $R_{GND}$  and  $R_{Vdd}$ ) for charging and discharging the capacitive load are assumed infinite.
- 2) In evaluating a network stage, transistors that are gated by nodes in state X are assumed to have a resistance represented by an interval and do not terminate the stage evaluation. This interval is included only in the Thevenin state evaluation and not in the resistance values used for estimating the delay times.
- 3) Recalling that an explicit estimate of transition times to the X state are not made. The transition time is defined to be the minimum of  $[t_{plh}, t_{phl}]$ .

There are two quite different interpretations of X states. One is to consider it as some intermediate voltage, say 2.5V. This is inconsistent with condition 2) because some contribution to the delay time would be made with this as the gate voltage. Within the RNL model, X is best considered as an undefined voltage. Condition 2) is then a very conservative statement of what these undefined nodes contribute to delay times.

##### 4.1.1. NMOS and CMOS Inverters

The effects of these conditions are highlighted by the propagation of an X state through NMOS and CMOS inverters. In both cases the state calculation reaches the correct answer that the output should also make a transition to X. The transition times for the two cases are now considered separately.

- For a NMOS inverter the transition time to logic H is independent of the inverter input (i.e. it uses a depletion pullup) and it reports this as the transition time  $R_{Vdd} \cdot C_{load}$ .
- In the CMOS case however, both logic H and logic L transition times depend upon the inverter input. Then from condition 2), in CMOS no contributions are made to lower the transition time from infinity. This leads to a rather unrealistic estimate for the delay time on the output.

On a node by node basis RNL does provide the capability to override the RC time constant. By explicitly setting the rise and fall times for a node, transitions to X are propagated with minimum of the two values. By its nature this solution removes one of the attractive features of RNL, the dynamic evaluation of signal delay times. Moreover, in the cases where there are more inputs to the output node (e.g. a nand) this effect can be difficult to track down. Let a word to the wise be sufficient.

#### 4.2. Node Overdrive

Important considerations for the following discussion;

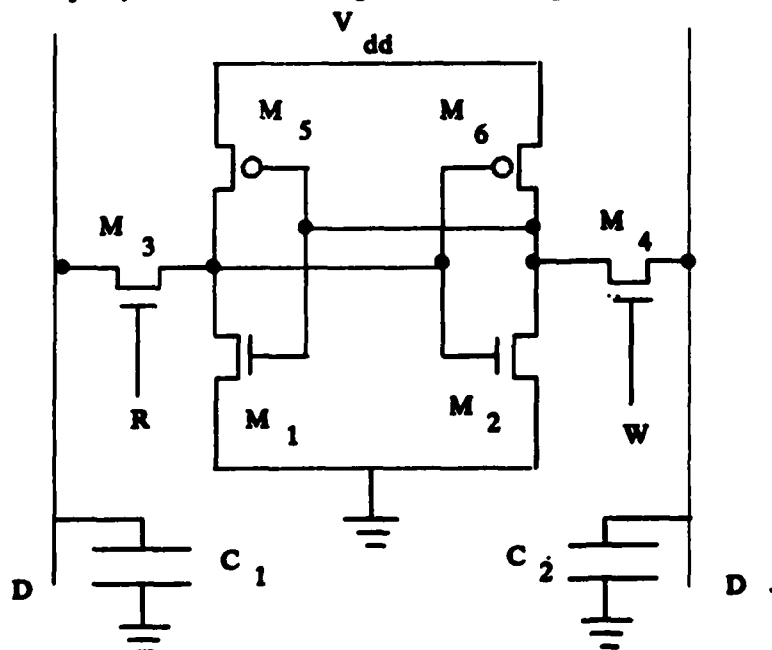
- 1) The replacement of transistors with resistors is independent of the logic thresholds declared for the transistor terminals.
- 2) Node states are determined by comparison of Thevenin resistance ratios with node logic values  $V_{low}$  and  $V_{high}$ .

When two (or more) charging elements are driving a single node, accurate modeling of node overdrive guarantees the right element wins. Node overdrive is used more frequently in CMOS design and should be of particular interest to those designers. Using this property in a digital circuit introduces significant dependence on analog properties of the circuit elements and nodes. In such cases it is suggested that an analysis tool such as SPICE be used to characterize the behavior of the subcircuit. With the SPICE results one can use several equivalent approaches to modeling the subcircuit with RNL. The following approach is trail and error but does not require that the device sizes be changed from those analyzed by SPICE.

From the theory of operation section of this user's guide recall that the final state of a node is determined by comparison of the Thevenin resistance ratios to parameters  $V_{low}$  and  $V_{high}$ . Furthermore, these parameters can also be set on a node by node basis. Values can then be found that reproduce the SPICE behavior for the node of interest. Depending upon the translation from transistors to resistors, the range  $V_{low}$  to  $V_{high}$  can be quite narrow. Again success of any of these methods depends on the tolerances in the design and all can be made to fail.

##### 4.2.1. Memory cell

An example of node overdrive that is commonly found in NMOS and CMOS designs is the 6 transistor memory cell. Only the CMOS example is shown here but the technique presented here works equally well for the analogous NMOS design.



We include from Hodges and Jackson ("Analysis and Design of Digital Integrated Circuits," D. A. Hodges and H. G. Jackson, McGraw-Hill Book Co., New York, p. 380.) a discussion on the sizing of the transistors in such a memory cell.

*To read a 1, D and Dbar are initially biased at about 3V. When the cell is selected, current flows through  $M_4$  and  $M_2$  to ground and through  $M_5$  and  $M_3$  to D. The gate voltage of  $M_2$  does not fall below 3V, so it remains on. However, to avoid altering the state of the cell when reading, the conductance of  $M_2$  must be about three times that of  $M_4$  so that the drain voltage of  $M_2$  does not rise above  $V_T$ . The operations of writing and reading a 0 are complementary to those just described.*

Such conservative design style should provide the designer with a working circuit without appeal to detailed analysis. Optimization, however, of the memory cell would be difficult to accomplish with RNL.

Simulating circuits of this type brings into focus an important conceptual difference between RNL and analysis tools. In RNL the logic threshold voltages  $V_{low}$  and  $V_{high}$  are declared independently from the replacement of transistors with resistors. In the memory cell this independence means that RNL could find transistor sizes that predict correct behavior for any set logic threshold voltages. Design of the 6 transistor memory is by no means the only case where the independence of the logic thresholds and transistor replacement can provide misleading results. As a rule subcircuits with analog properties should be analyzed using other tools first and then RNL parameterized to fit that behavior.

#### 4.3. Floating Nodes

Important considerations for this discussion are;

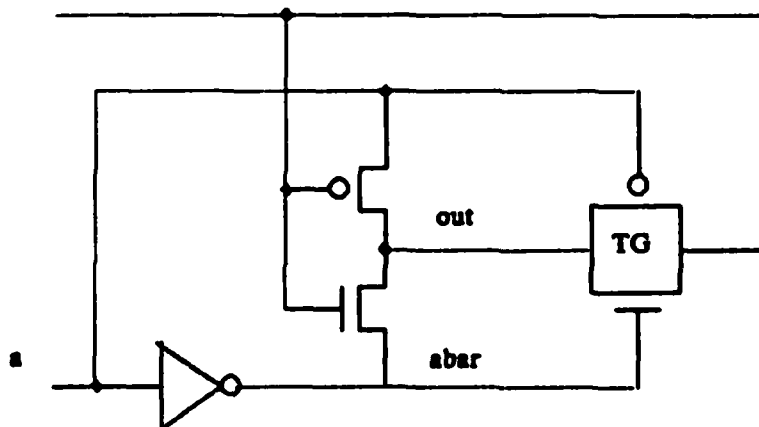
- 1) The effects of a charge sharing event are evaluated immediately.
- 2) Only one charge sharing and one final state event can be pending for each node in the circuit.

Floating nodes present another case where the analog properties dominate the prediction of the behavior of the subcircuit. In many cases a floating node can have several inputs making it difficult to find a set of RNL parameters for the nodes involved. The following example demonstrates these difficulties.

##### 4.3.1. Exclusive or

This is a CMOS design of an exclusive or where all transistors can be of minimum size. It requires the transmission gate (TG) to compensate for the poor transmission of a logic low by a p type enhancement and conversely logic high by a n type enhancement.

b



Initially it was thought that *out* would require the most attention in the parameterization. Naive RNL simulation provided reasonable predictions for the cases where TG contributed to the final state of *out*. The states that utilize the TG are indicated in the state table below. RNL did not predict proper behavior in other cases. For example, when *b* was high and *a* changes from low to high, *out* was predicted to be X. This was not the result of SPICE analysis. It was also found that correct predictions for the other states could only be obtained when nodes *a* and *b* were set at the same time in the simulation. Another alternative was to declare *abar* as an input. This evidence suggests that



the other nodes (e.g. *a*, *abar*, *b*) should be the focus. Why?

a	b	out
0	0	1 (TG)
0	1	0 (TG)
1	0	1
1	1	0

In the first case the simultaneous transition of *a* and *b* produce the proper final state because the result of charge sharing is the correct final state. Secondly, declaring any node an input (in this case *abar*) provides it with zero impedance for that state. For the duration of it being an input, this excludes any possible state changes including the offending charge sharing event.

The difficulty in simulating this circuit with RNL centers around the fact that the node *out* is controlled by the nodes that are also its inputs, namely *a* and *abar*. Transitions of either node can promote the scheduling of both charge sharing and final state transitions. The effects of the charge sharing event are by design evaluated immediately. Without careful consideration of the  $V_{low}$  and  $V_{high}$  for the nodes *a* and *abar*, the charge sharing events can result in a final state of X for node *out*. The correct final state for *out* is then discarded by RNL (only one charge sharing and final state change can be pending) and is replaced by a transition to X.

#### 4.3.2. Summary

From the examples presented RNL, used with some care, does have the ability to reproduce many of the results obtained from other analysis tools. Due to the independence of transistor replacement and node logic voltages detailed analysis of subcircuits should be done when analog properties dominate the subcircuits behavior.

#### 4.4. Problems With Circuit Initialization.

The functions *sim-init* and *switch-init* operate by walking through the network and for each node that is in the X state scheduling on the event queue an immediate transition of that node to the 0 state. A subsequent advancement of the simulated time will allow these transitions to occur and their effects to propagate. This is especially useful in circuits that contain storage elements that cannot be controlled by the inputs to the circuit. A typical example is a divider circuit composed of a chain of latches and whose only input is a clock line. Because RNL interprets X as the "unknown" rather than as the "intermediate" state, a flip-flop which contains X's will remain "unknown" rather than go through a transition from a meta-stable to a well-defined state. (Circuit analysis simulators such as SPICE have their own problems when it comes to resolving meta-stable states.)

While it is sometimes necessary to use *sim-init* or *switch-init*, there are cases in which these functions will be unreasonably expensive, taking hours or even days to initialize the circuit. This is especially true if *sim-init* is called before any other attempts are made to initialize the circuit. This phenomenon has two related causes:

*Sim-init* schedules its (nominally simultaneous) transitions in the order that it finds the nodes in the node hash table. This is in contrast to scheduling nodes closer to inputs first, or other related schemes. This can result in a lot of extra events being scheduled and evaluated. A typical example is when an output of a NOR gate is initialized before its inputs. The effects of the output being in a 0 state are computed and propagated even though the evaluation of the inputs will eventually put the output in a 1 state.

The evaluation of events during initialization can be very, very expensive compared to comparable events during the normal operation of the circuit. The reason for this is again the interpretation of the X state. When RNL attempts to evaluate the effects of an event on some node A, it examines the states of all the other nodes connected to A. Since the interpretation of X is "unknown", a transistor whose gate is X might be on. If A is the source or drain of that transistor then it might affect the final state of A and therefore the computation must consider this. If the node A happens to be affected by many transistors (consider one line in a bus), and if the control lines gating those transistors are in state X, then each time any of the nodes connected to the bus through one of these "maybe on" transistors goes through a transition all of the nodes will be examined. This is exactly what happens when `sim-init` is invoked before the control lines are initialized. Eventually the control lines may be initialized, but by then the damage has been done.

The moral is that one should be careful when one attempts to initialize a circuit in RNL. Good design practice dictates that for testability purposes it should be easy and efficient to put a circuit into some known state by driving its inputs. One should attempt to use that initialization protocol to initialize simulated circuits also. The cases in which this does not seem to work are those in which there are embedded state machines that must already be in a well-defined state before they can be initialized. If the outputs of these sub-circuits control large parts of the chip then a special initialization protocol for these parts should be considered. Only after one has initialized what can be controlled from inputs and only after one has eliminated the X's on the major control lines of the circuit should one consider using `sim-step` to do the residual initialization.

## 5. USER INTERFACE

The user interface of RNL is a simple LISP interpreter. This is a brief introduction to that version of LISP.

The interpreter continually executes the following loop:

- (1) read a command from current input;
- (2) evaluate the command, performing the specified actions;
- (3) print the result and loop back to (1).

There are two syntactic forms for specifying commands to this loop. The most general looks like

```
(function argument argument ... argument)
```

i.e., a list of names, numbers, etc. separated by white space (spaces, tabs, and newlines) and enclosed in parentheses. The parentheses delimit the command, so that the white space can be used to format the input any way one pleases. The arguments themselves may also be of the form (function arg ... arg). The interpreter first reads the entire command -- up to the closing parenthesis. The first element of the list is interpreted as a function. The arguments are then evaluated in left-to-right order and the results passed to the function. The value returned by the function is printed and the reader invoked once again. For example, given the following input

```
(* 17 (+ 3 2)
(/ 10 2))
```

RNL would respond by typing 425 and then wait for more input. Note that nothing happened after the first newline since the first parenthesis had not yet been closed.

The reader for the command interpreter also accepts commands of the form:

```
function argument argument ... argument < newline>
```

This is equivalent to

```
(function '(argument argument ... argument))
```

The `'` is shorthand for the quote special form. This keeps its argument from being evaluated. Quote is explained in more detail below. Many of the simple simulator functions contained in the file `"uwsim.l"` are written this way in order to eliminate the typing of parentheses when invoking common

commands.

Comments can be included by preceding them with a semicolon (;). All characters following the ";" up through the next newline are ignored.

### 5.1. Objects and Values

The RNL LISP interpreter allows you to access the following types of objects:

**numbers** -- signed integers. (16 bits on PDP11s, 24 bits on VAXen, 28 bits on PDP10s).  
 -- floating point. (the standard single precision format for the machine).

**strings** sequence of characters enclosed in quotes ("). Useful as constants for file names, print statements, etc. Special characters can be introduced into the strings by using the backslash escapes:

'\n' newline

'\r' return

'\t' tab

'\ooo' ascii code "ooo" where ooo are octal digits

**symbols** are like variables in other programming languages. A symbol is referred to by its *print name*: any sequence of characters (not including a period ".") delimited by "white space" that isn't a number or string. Special symbols (including white space and control characters) can be included in symbol names by using the backslash escape convention. Long symbol names with embedded blanks and all other special characters can be created by enclosing the name in a pair of vertical bars.

Example: | long symbol\014 name | defines a

**nodes** long name with a form feed in the middle. Use long symbols in preference to strings.  
 are the electrical nodes of your circuit. Although they may have names that resemble symbols, they are a distinct data type. Note that many nodes have print names that are numbers. Symbols and numbers are distinguished from nodes by the context in which they are used. In addition to numbers and symbols, nodes can have structured names of the form "a.b.c. ..." where each of a is a symbol and b, c, etc are symbols or numbers. This allows you to create arrays and hierarchical naming schemes for your nodes. It has the unfortunate side effect of forcing you to use the vertical bar convention to enter symbol names containing periods, i.e. |a.b.c|.

**lists** are sequences of objects enclosed in parentheses. Standard LISP syntax applies, including dot notation. The empty list "()" is also called "nil".

**subrs** primitive, or built-in, functions (like +).

One can evaluate an object for a value; numbers, strings, subrs, and nodes are "self-evaluating", i.e., the object and its value are one and the same.

Evaluating a symbol yields the value last assigned to that symbol by the user (see the setq function). Symbols actually have two distinct values: the value used during evaluation and one used only when the symbol is used as a function name. A useful example of this is the symbol "\*" which when used as a function denotes multiplication, but which when used as an argument denotes the last value returned by a command to the top level interpreter.

Evaluating a list is like making a function call. The function value (or the ordinary value if there is no function value) of the first element of the list is the function. The values of the remaining list elements are the arguments. For example, evaluating

`(+ a 3)`

looks up the function value of the "+" symbol (in this case it will be the subr for addition), then calls the function with the values (recursively computed) of "a" and "3". The value of the list will be the value returned by the function.

Certain lists have special meaning to the system and are called special forms. Two special forms of particular interest are discussed here, the remainder are described in a later section. The quote special form,

`(quote arg)` or `'arg`

allows us to create symbol and list constants. Thus the value of `(quote a)` is the symbol "a", and the value of `'(+ 2 3)` is a list of three elements.

User defined functions are represented by the lambda special form:

`(lambda (param param ...) exp exp ...)`

The symbol "lambda" indicates that this list is actually a user function. It is followed by a list giving the names of the arguments and finally by a sequence of expressions which make up the body of the function. The value returned by the function when called will be the value of the last expression in the body. For example,

`((lambda (x) (+ x 3)) 4)`

evaluates to 7. We can give this function a name by making the lambda expression the value of some symbol:

`(setq plus-3 '(lambda (x) (+ x 3)))`

`(plus-3 4)`

also evaluates to 7. In doing this we set the ordinary value of plus-3 to the lambda expression. A better way of doing this is to use

`(defun plus-3 (x) (+ x 3))`

which makes `(lambda (x) (+ x 3))` the function definition of the symbol "plus-3".

Note that `setq` changes the "expression value" of a symbol, while `defun` changes the "function value" -- this distinction is unimportant in most applications, but is useful if you wish to change the definition of a built-in function (beware of the implications before trying to change built-in function definitions though!).

This version of *rnl* is case sensitive. Special nodes with names "Vdd" and "VDD" are aliased to "vdd"; "Gnd" and "GND" are aliases of "gnd".

## 5.2. About Efficiency

LISP symbols and circuit nodes with the same name are in fact *different* objects. Functions that take nodes as arguments also work with LISP symbols, but the symbol is converted to a node each time the function is called. This entails getting the print name of the symbol and using it as the key in a hash table lookup of the node. While this is implemented efficiently, it is still much more expensive than using the node directly. Often used arguments should therefore be converted from symbols to nodes using `find-node`. (This conversion can be done only after the network has been loaded because the nodes are not created until then.)

## 5.3. Useful Symbols

The following symbols are defined by RNL and are accessible to the user. They are useful in writing your simulations.

- is the value last returned by the top level loop. This is mostly useful when you are poking around interactively.
- base controls the radix for printing integers. If not one of 2, 8, 10, or 16 then base 10 is used. The input radix is controlled by using the Unix conventions extended by using "0B" to signal a binary integer.

**current-time** is the current value of simulated time, expressed in tenths of nanoseconds.  
**end-of-file** returned when EOF is read.  
**event-list-empty** is set to *t* when there are no pending electrical events on the queue, all otherwise.  
**user-interrupt** is set to *t* if the user types the "quit" character. (Default is ctrl-\.) A check for **user-interrupt** as a condition for exiting a time consuming loop is often useful.

## 6. BUILT-IN FUNCTIONS

The file "uwstd.l" contains functions that are usually found in LISP environments. This section documents these functions in addition to the functions that are really built in to RNL. The functions are grouped by application area. The areas are:

arithmetic functions and predicates

list and symbol manipulation functions

i/o functions

special forms

network/simulation functions

Unless otherwise stated all functions evaluate their arguments. In addition to standard functions described in this section, you have the option of loading the file "uwsim.l" which contains a suite of functions that implement a collection of useful and relatively easy to use "front-end" facilities for doing circuit simulations.

The notation used in the descriptions of the functions is intended to make clear the types of the arguments. Arguments are prefixed as follows:

*g\_* for any type,

*s\_* for a symbol,

*t\_* for a string,

*p\_* for types with unique print names (symbols, nodes, strings, and integers),

*c\_* for symbols and nodes,

*l\_* for list arguments,

*n\_* for any number,

*i\_* for integer,

*f\_* for floating point.

Quoting the formal parameter means that the argument evaluates to the required type. Special types are mentioned explicitly. For example (func '*i\_arg*) means that *i\_arg* evaluates to an integer and (func '*vec*) means that func takes an argument that evaluates to a circuit node vector.

### 6.1. Arithmetic functions

Unless otherwise stated, the arithmetic functions take both floating point and integer arguments, returning a floating point result if any argument was a floating point number. Warning: overflow and underflow are not checked by these functions.

( \* 'n\_arg1 ... 'n\_argn )

returns the product of its arguments.

( + 'n\_arg1 ... 'n\_argn )

returns the sum of its arguments.

( - 'n\_arg1 ... 'n\_argn )

returns the first argument minus the sum of the remaining ones. The form ( - n\_arg ) returns the negation of its argument.

( / 'n\_arg1 'n\_arg2 'n\_arg3 ... )

returns the quotient of n\_arg1 divided by the product of the rest of the arguments. If all arguments are integers the result is an integer truncated towards zero.

( % 'i\_arg1 'i\_arg2 ) returns the remainder of i\_arg1 divided by i\_arg2.

( 1+ 'i\_arg )

like ( + 'i\_arg 1 ) but restricted to integers.

( 1- 'i\_arg )

like ( - 'i\_arg 1 ) but restricted to integers.

( < 'n\_arg1 'n\_arg2 )

returns t if n\_arg1 less than n\_arg2.

( <= 'n\_arg1 'n\_arg2 )

returns t if n\_arg1 less than or equal to n\_arg2.

( == 'n\_arg1 'n\_arg2 )

returns t if n\_arg1 (numerically) equal to n\_arg2.

( != 'n\_arg1 'n\_arg2 )

returns t if n\_arg1 (numerically) not equal to n\_arg2.

( > 'n\_arg1 'n\_arg2 )

returns t if n\_arg1 greater than n\_arg2.

( >= 'n\_arg1 'n\_arg2 )

returns t if n\_arg1 greater than or equal to n\_arg2.

( abs 'n\_arg )

returns the absolute value of n\_arg.

( fix 'n\_arg )

returns integer part of *n\_arg*.  
truncated towards zero.

(float '*n\_arg* )

returns floating point version of *n\_arg*.

(max '*i\_arg1* ... '*i\_argn* )

returns maximum of its (integer arguments.

(min '*i\_arg1* ... '*i\_argn* )

returns minimum of its (integer) arguments.

(numberp '*g\_arg* )

returns t if *g\_arg* is a integer, nil otherwise.

## 6.2. Functions and Predicates for List and Symbol Manipulation

(alphalemp '*p\_arg1* '*p\_arg2* )

returns t if *p\_arg1*'s print name is lexicographically less than *p\_arg2*'s.

(append '*l\_arg1* '*l\_arg2* )

returns list of *l\_arg1* with *l\_arg2* appended to it. (This is defined in "uwstd.l".)

(atom '*g\_arg* )

returns t if *g\_arg* is not a list.

(car '*l\_arg* )

returns first element of list *l\_arg*.

(cdr '*l\_arg* )

returns list of all but first element of *l\_arg*.

(c\*r '*l\_arg* )

equivalent to multiple car and cdr (\* = aa,ad,da, or dd). (This is defined in "uwstd.l".)

(char-to-num '*p\_arg* )

returns the ASCII code for the first character of *s\_arg*'s print name.

(cons '*g\_arg1* '*g\_arg20* )

returns a list *l* such that (car *l*) = *g\_arg1* and (cdr *l*) = *g\_arg2*.

(eq '*g\_arg1* '*g\_arg2* )

returns t if *g\_arg1* and *g\_arg2* are the same (identical !!) LISP object.

(equal '*g\_arg1* '*g\_arg2* )

returns t if *g\_arg1* and *g\_arg1* are conformable.

( *explode* '*p\_arg*' )

returns a list of symbols whose single character names are the characters of *p\_arg*'s print name.

( *fset* '*s\_arg*' *lambda* )

sets *s\_arg*'s function definition.

( *fsymeval* '*s\_arg*' )

returns *s\_arg*'s function definition.

( *get* '*s\_arg*' *g\_name* )

returns value of *s\_arg*'s *g\_name* property. (This is defined in "uwstd.l".)

( *implode* (*p\_arg1* *p\_arg2* ... ) )

inverse of *explode*. Arguments with no sensible print name are ignored.

( *length* '*l\_arg*' )

returns number of elements in list *l\_arg*.

( *list* '*g\_arg1*' *g\_arg2* ... )

makes a list with elements *g\_arg1*, etc.

( *make-symbol* '*p\_arg1*' *p\_arg2* ... )

returns symbol whose pname is concatenation of pnames of its arguments. This is very useful for converting nodes to symbols. See *implode*.

( *mapcar* '*u\_func*' '*l\_arg*' )

returns a list whose elements are the result of applying *u\_func* to each of the elements in *l\_arg*. (This is defined in "uwstd.l".)

( *memq* '*g\_arg*' '*l\_arg*' )

returns tail of *l\_arg* beginning with *g\_arg*, if *g\_arg* is not in *l\_arg*, then it returns *nil*. This uses *eq* to test equality.

( *null* '*g\_arg*' )

returns t if *g\_arg* is *nil*. *not* is a synonym for *null*.

( *plist* '*s\_arg*' )

returns *s\_arg*'s property list. (This is defined in "uwstd.l". It is not intended to be used directly by users. See "get".)

( *plist* '*s\_arg*' '*l\_arg*' )

sets *s\_arg*'s property list to *l\_arg* and returns *l\_arg*.

(This is defined in "uwstd.l". It is not intended to be used directly by users. See "putprop".)



- ( **pname** *'p\_arg* )  
returns string equal to *p\_arg*'s pname.
- ( **putprop** *'s\_arg* *'g\_val* *'g\_name* )  
sets *s\_arg*'s *g\_name* property to *g\_val* and return *g\_val*. (This is defined in "uwstd.l".)
- ( **remprop** *'s\_arg* *'g\_name* )  
returns *s\_arg*'s property list from *g\_name* on and removes *g\_name* property from list. (This is defined in "uwstd.l".)
- ( **rplaca** *'l\_arg* *'g\_arg* )  
replaces car of *l\_arg* with *g\_arg*.
- ( **rplacd** *'l\_arg* *'g\_arg* )  
replaces cdr of *l\_arg* with *g\_arg*.
- ( **set** *'s\_arg* *'g\_arg* )  
sets value of *s\_arg* to *g\_arg* and returns *s\_arg*.
- ( **setq** *s\_arg* *'g\_arg* )  
sets value of *s\_arg* to *g\_arg* and returns *s\_arg*.
- ( **stringp** *'g\_arg* )  
returns t if *g\_arg* is a string.

### 6.3. I/O functions

The notation [*fid*] indicates an optional file identifier argument. If it is included the operation is directed to the designated file. If omitted the operation goes to the standard input or output device as appropriate.

The strings used to specify filenames can contain all the standard UNIX file name expansion conventions including the use of environment variables (e.g. ".../myfile").

The base used for printing integers is controlled by the value of the symbol "base". The default is decimal, base 10.

- ( **close** *'fid* )  
close file specified by *fid*.
- ( **flush** *'fid* )  
force buffered output for file *fid*.
- ( **load** *'p\_name* )  
take input from file named by *p\_name*.  
If the file is not found in the present working directory then the directories specified in the environment variable RNLPATH will be searched. If RNLPATH is not defined then the default directory \$UW\_VLSI\_TOOLS/lib/rnl is searched. UW\_VLSI\_TOOLS is an environment variable that is used in many of the tools distributed by UW/NW VLSI Consortium. This searching is done only for load.

**( log-file '*p\_name*' )**

closes the currently open log file (if any) and opens a log file named *p\_name* and returns a *fid* for the file. (log-file nil) closes the currently open log file. A log file contains a verbatim copy of everything that goes to your terminal during the time that it is open.

**( openi '*p\_name*' )**

open file with name *p\_name* for input, return *fid* for the file opened.

**( openo '*p\_name*' )**

open file with name *p\_name* for output, return *fid* for the file opened.

**( prin1 '*g\_arg*' [*fid*] )**

prints *g\_arg* without trailing newline.

**( princ '*g\_arg*' [*fid*] )**

like prin1 without quotes around strings.

**( print '*g\_arg*' [*fid*] )**

print *g\_arg* followed by newline.

**( printf [*fid*] '*string*' '*g\_arg1*' '*g\_arg2*' ... )**

print *g\_arg1*... under format control specified by *string*. This is similar to the printf in the `stdio` library for C. Escapes for printing the *g\_arg*'s are: %c-> ASCII char, %%-> print '%', and %S-> print LISP object.

```
Example: (setq a 10)
          (setq b '(list of symbols))
          (printf "a=%d0=%S0 a b)
produces
a=10
b=(list of symbols)
```

**( read [*fid*] )**

read an S-expression from an appropriate file. Returns the expression read or the symbol `end-of-file` if the end of the file is reached. This does not recognize the shorthand syntax of the standard read-eval-print loop.

**( read-network '*p\_name*' )**

read a network in file named *p\_name*.

This file must be the output of PRESIM. The network described in the file is merged with the networks already loaded. There is no way to undo the loading of a network other than restarting RNL.

**( read-state '*p\_name*' )**

reset state of circuit to that in file named *p\_name*.

( **terpri** [*fid*] )

output newline.

( **write-ascii-state** '*p\_name* )

save readable form of current state in file named *p\_name*. This cannot be read back by **read-state**.

( **write-state** '*p\_name* )

current state in file named *p\_name*. This can be read by **read-state** routine.

#### 6.4. LISP Control Structures and Special Forms

Although special forms look like function calls, their behavior can be quite different (especially with respect to the evaluation of their "arguments").

**Warning:** Lambda bound variables (parameters to **defun**, **lambda**, or **prog**) remain bound until the form is exited. Functions called from within these forms will see the new bindings. This is not consistent with other dialects of LISP.

( **and** *g\_exp1 g\_exp2 ...* )

evaluates *g\_exps* left to right until the first nil result is found. And returns the value of the last argument it evaluates.

( **cond** *clause clause ...* )

Following the **cond** is a sequence of clauses. Each clause is of the form ( *pred exp exp ...* ). In each clause the first expression is taken to be a predicate and the remainder of the expression the body of the clause. **Cond** evaluates the predicates in turn, stopping at the clause with the first non-nil value for the predicate. The body of that clause is then evaluated and the value of the last expression in the body is returned as the value of the **cond**. If no predicates evaluate to non-nil, nil is returned as the value of the **cond**.

( **defun** *s\_name (s\_par ...) exp ...* )

define function *s\_name* with formal parameters *s\_par ...*. This is equivalent to ( **let** '*s\_name* '( **lambda** (*s\_par ...*) *exp ...* ) ).

( **do** (*l\_vrbl ...*) *l\_exit1 exp1 exp2 ...* )

is a generalized iteration expression.

It has three components:

- (i) a list of iteration symbol declarations,
- (ii) an exit clause,
- (iii) and a body that is executed on each iteration.

The list of iteration symbol declarations are used to declare temporary (i.e. "prog" bound) symbols whose scope is the **do** expression. Upon exiting the **do** they revert to their previous status. Each declaration in the list has one of the forms:

(*s\_sym*) or (*s\_sym* '*g\_init*) or (*s\_sym* '*g\_init* '*g\_iter*)

where *s\_sym* is the symbol declared, '*g\_init*' is an expression whose value is assigned to *s\_sym* at the start of the first iteration, and '*g\_iter*' is an expression that is evaluated at the start of each successive iteration to provide successive values for *s\_sym*. If '*g\_iter*' is omitted then the value is not changed automatically. The initialization and iteration expressions are evaluated in left-to-right order. For example, the following declaration list says that *i* starts at zero and is incremented by two, and that *j* starts at 3 more than the value of symbol *k* and is squared on each iteration.

```
((i 0 (+ i 2)) (j (+ k 3) (* j j)))
```

The exit clause, *l\_exit*, is a list of the form

```
(pred eexp1 eexp2 ...)
```

After the new values have been assigned to the iteration symbols, the predicate *pred* is evaluated. If its value is non-nil, then the *eexp*'s are evaluated in order and the value of the last is returned as the value of the *do*. If the value of *pred* is nil, then the body of the loop, i.e. expressions *exp1*, *exp2*, ..., is evaluated. This process is repeated until *pred* is non-nil.

```
(eval 'g_arg)
```

evaluates *g\_arg* and returns the result.

```
(exit)
```

exits *rnl* in an orderly fashion, flushing buffers, closing files etc.

```
(lambda (s_par1 s_par2 ...) exp1 exp2...)
```

is the definition of a (nameless) function with formal parameters *s\_par1 s\_par2 ...*. *lambda* itself is not a function.

```
(or g_exp1 g_exp2 ...i)
```

evaluates the *g\_exps* left to right, stops when the first nil is returned. Or returns the value of the last argument if it evaluates.

```
(prog (s_1 s_2 ...) 'g_exp 'g_exp ...)
```

saves the old values of *s\_n* s, binds the symbols to nil, and evaluates each of the *g\_exps* in order. The old values are restored when execution of the *prog* is completed. Returns the value of the last of the *g\_exps*. *Prog* is used to allocate local variables within a function.

```
(quote g_arg)
```

returns *g\_arg* without evaluating it. The syntax '*g\_arg*' is equivalent.

```
(repeat s_index 'l_lower 'l_upper expr_list)
```

is a simple iteration similar to a *for* loop in Pascal. *Expr\_list* is the body of the loop. *S\_index* is the index variable. *l\_lower* and *l\_upper* are integers representing the initial and the final values of the index. The index is increased by one each time the loop is executed. Returns the final value of the index.

## 6.5. Network functions

An electrical network consists of a list of nodes transistors, capacitors, and resistors. The functions described in this section allow user-defined functions to deal with the network.

**( l 'l\_nodes )**

prints a list of the transistors whose gates are connected to nodes in 'l\_nodes.

The syntax: l node1 node2 ... is more common. Returns nil.

**( ? 'l\_nodes )**

prints a list of transistors whose source or drain are connected to nodes in l\_nodes. The syntax: ? node1 node2 ... is more common. Returns nil.

This command is useful for wandering through the network trying to track down the source of a particular value.

**( find-node 'p\_arg )**

returns electrical node with print name the same as p\_arg's. Returns nil if there is no such node.

**( match-node 'p\_pattern )**

uses p\_pattern's print name as a pattern using '\*' as a wild card. Returns a list of symbols whose print names correspond to print names of nodes that match the pattern.

**( node-value 'p\_arg )**

if a node exists with a print name matching p\_arg's return its value (one of 0, 1, or X), otherwise nil.

**( set-delay 'c\_node 'i\_tplh 'i\_tphl )**

Set the transition times for the specified node; tplh (low-to-high transition time) and tphl (high-to-low transition time) are integers specifying time in 10ths of nanoseconds. If either tplh or tphl are negative, the node's times become unspecified and the transition times will be determined by the usual RC calculation. This command allows one to override the timing calculation. This is useful when the RC calculation gets the wrong answer for one reason or another. Usually this is worth doing only on critical nodes, such as clocks, where an timing error can be significant.

**( set-node 'c\_node 'g\_exp )**

set value of node c\_node to g\_exp. adds/removes node as an input. If exp evaluates to

0	node is added to low input list
1	node is added to high input list
U,u	node is added to undefined input list
X,x	...see text...

The node will be stuck at this input value until changed by another call to set-node. If exp is X (remember exp is evaluated so you'll probably want to type 'X'), node is removed from the input lists. At the next simulation step it will acquire whatever value it would naturally have.

**( set-params name value )**

give a value to one of the simulation parameters:

<b>report</b>	flag (default = t). If non-nil, nodes given the value of X because of improper pullup/pulldown ratio or because of charge decay will cause a warning message to be printed.
<b>unitdelay</b>	flag (default = nil). If non-nil, all node transitions happen with unit delay.
<b>decay</b>	fixnum (default = 0). If non-zero, tells the number of time units (10ths of nanoseconds) it takes for charge on a node to decay to X. A value of 0 implies no decay at all.
<b>maxres</b>	number (default = 1E10). Capacitors on the far side of transistors bigger than this value don't contribute to summary capacitance used in calculating transition times.

( **set-threshold** 'c\_node' f\_vlow f\_vhigh )

set *c\_node*'s logic thresholds to *f\_vlow* (low) and *f\_vhigh* (high). *vlow* and *vhigh* should be numbers in the range [0,1]. These thresholds are used when converting from a Thevenin equivalent voltage to a logic state -- sometimes it is useful to be able to override the defaults for special nodes which otherwise will turn out X.

( **sim-init** )

This finds all nodes whose values are X and queues a transition to 0 for those nodes. The integer returned is the number of affected nodes. A call to **sim-step** or one of the higher level simulation commands such as "step", "s", or "c" will allow these changes to propagate. Initialization should be done by manipulation of the inputs of the circuit, simulating the real initialization sequence. **Sim-init** can be used to initialize nodes that cannot otherwise be initialized. Using **sim-init** without first simulating the setting of the inputs can be very, very expensive, especially when trying to initialize circuits connected by a bus.

( **sim-step** 'i\_stop-time' )

simulation step using RC model. This runs the electrical simulation until *current-time* = *i\_stop-time*.

If the simulation runs to completion then nil is returned. If a node that has the STOPON-CHANGE flag set goes through a transition, then the simulation is stopped at that point and the node that had the transition is returned.

( **stop-on-change** 'c\_node' g\_switch )

If *g\_switch* is not nil then set *c\_node*'s "STOPONCHANGE" flag. If *g\_switch* is nil then *c\_node*'s "STOPONCHANGE" flag is cleared.

( **switch-init** )

like **sim-init**, except prepares network for initialization by **switch-step** instead of **sim-step**.

( **switch-step** 'stop-time' )

simulation step using switch model. This is similar to **sim-step**, but transistors are modelled as switches and transitions have unit delay. This algorithm is somewhat faster than the usual RNL calculation for many circuits, but can give X answers for circuits for which transistor size is important for correct operation (e.g., bit line in a dynamic memory). To ensure correct operation, one should not use **sim-step** until the event list is empty (and vice versa) -- i.e., all events scheduled by a particular algorithm must be handled by the same algorithm. The value of

**event-list-empty** can be tested to see if the all events have been handled.

This routine may be useful when debugging the basic functionality of a circuit, or when simulating a circuit which has not been correctly sized (one that gives ratio errors using *sim-step*). Since the switch-level algorithms are much faster when dealing with large groups of interconnected nodes, *switch-step* may be particularly useful when initializing a network.

**( walk-net 'function )**

*function* should be a symbol or a lambda expression that takes a circuit node as its only argument. *walk-net* applies that function to every node in the network.

**( trace-node 'c\_node 'g\_switch )**

If *g\_switch* is not nil then start to trace *c\_node*, otherwise stop tracing *c\_node*. This is useful for trying to track down exactly what is happening to a subcircuit at a very low level. The first form turns on tracing for the specified node, the second turns it off. Sample outputs:

```
[1]; event 1: b=H @ 10.0ns
[2]; b => clist: d < input seen >
[3]; d: rgnd=[3.05e+04,6.11e+04], rvdd=[1.00e+10,1.00e+10]
[4];      d-rgnd=1.18e+04, d-rvdd=1.00e+10, lhdelay=0, hldelay=0
[5];      cap: high=0.000000e+00, low=0.000000e+00, x=1.014720e+00
[6];      => value=L @ 1.20e+01 ns
[7]; enqueing d [event 1: b] L @ 220 (delta = 120)
```

[1]: node b makes a transition to H at 10.0ns

[2]: a list (clist) of nodes affected by b is reported. In this case one node (d) is found before an input ends the search. Inputs can be forced nodes, vdd or gnd.

[3]: Report the result of the Thevenin calculations for nodes on the clist, rgnd Thevenin resistance to ground, rvdd Thevenin resistance to vdd. Note in this first case rgnd is computed to be an interval. This is the result of an input node having a value of X.

[4]: Report the value of the resistors to gnd (d-gnd) and vdd (d-vdd) used in the delay calculations. Note these values are not necessarily the same as those in [3]. This is the result of using different values for  $R_{static}$ ,  $R_{dynlow}$  and  $R_{dynhigh}$ .

[5]: Report the value of total capacitance charged high, low and x for current node.

[6,7]: Compute new logic value for node and enqueue it at current-time + delta (delta = RC). R and C are chosen from the values given in lines [4] and [5].

```
[8]; event 1: c=H @ 10.0ns
[9]; c => clist: d < input seen >
[10]; d: rgnd=[1.00e+10,1.00e+10], rvdd=[3.05e+04,3.05e+04]
[11];      d-rgnd=1.00e+10, d-rvdd=5.90e+03, lhdelay=0, hldelay=0
[12];      cap: high=0.000000e+00, low=1.014720e+00, x=0.000000e+00
[13];      => value=H @ 6.00e+00 ns
[14]; enqueing d [event 4: c] H @ 531 (delta = 60)
```

This example is substantially the same as the first except that the Thevenin resistance is no

longer an interval.

( trace-all-nodes 'g\_switch )

If *g\_switch* is not nil then start to trace all nodes, otherwise stop the trace. Sometimes this is the only way to track down oscillating subcircuits.

## 7. The *uwsim.l* package.

The *uwsim.l* package is intended to provide a powerful and easy to use front end for *rnl*. In addition, it is can serve as the basis for customized front ends for specific projects. In this document we concentrate on the functions intended to be directly called by users. The programmer who intends to extend this or to customize it is invited to peruse the code.

### 7.1. Syntax and symbols.

The *uwsim.l* package defines two new data-structures: Vectors of circuit nodes are defined by *vecdef*'s. A *vecdef* is a list of the form ( *s\_mode s\_name c\_node1 ... c\_noden*). *s\_mode* is one of {hex, dec, oct, bin, bit} and controls the formatting of output and input for the vector. The first four modes allow numeric input in any base and output in the specified base (and are limited to 23 nodes), the bit mode uses bit vector input and output. If a numeric vector contains undefined nodes it is printed as a bit vector. The first element in the list of nodes is the high-order bit of the numeric vector.

You can request that a report about the state of your circuit be printed at certain times during the simulation (usually at the end of a clock cycle). The format of this report is specified by a report-form which is a list containing *vecdefs*, strings, symbols corresponding to nodes, nodes, and the format control symbols { newline, tab, and page} printing of a report. The car of the list is a string that heads the report.

It is useful to know about the following symbols:

<i>t</i>	The property list of "t" is used to hold tokens describing which packages have been loaded. <i>Uwsim.l</i> requires that <i>uwsim.l</i> be loaded first.
<i>incr</i>	is the time interval used by the step and clock functions. The default is 1000 (= 1000ns).
<i>relative-timing</i>	If this is not "nil" then transition times are reported relative to the start of the step.
<i>switch-level</i>	If "switch-level" is non- nil then the switch-level is used, otherwise the RC model.

### 7.2. Functions intended for Direct Use by Users.

( *defvec* 'vecdef )

optimizes the representation of *vecdef* by converting LISP symbols into nodes and stores that representation as the *vecdef* property of *vecdef*'s name.

( *def-report* 'l\_arg )

creates an optimized report form. *l\_arg* is a list, beginning with a title string, containing the following LISP forms: strings are printed in the report without surrounding quotes.



**newline** inserts a newline in the output.

**tab** inserts a tab in the output.

**page** inserts a form feed in the output.

**s\_arg** for symbols other than those above causes the state of the corresponding node (if any) to be displayed. An error is reported and a newline is inserted if *s\_arg* cannot be converted to a node.

**(vec s\_arg)**  
or  
**(vec vecdef)** causes the state of a vector to be inserted in the report. If the first form is used, *s\_arg* should have been previously been given a vector definition. If the second form is used, the vector definition is created on the fly.

**(function g\_arg)** causes *g\_arg* to be evaluated when this form is encountered in the printing of the report.

**( h 'l\_nodes )**  
makes the nodes in *l\_nodes* inputs at logical high (1). (Alternate syntax: h node1 node2 ...)

**( l 'l\_nodes )**  
makes the nodes in *l\_nodes* inputs at logical low (0). (Alternate syntax: l node1 node2 ...)

**( x 'l\_nodes )**  
removes the nodes in *l\_nodes* from the input list. They can now be driven high or low by the modelled circuit. (Alternate syntax: x node1 node2 ...)

**( u 'l\_nodes )**  
makes the nodes in *l\_nodes* to be undefined inputs. (Alternate syntax: u node1 node2 ...)

**( t 'l\_nodes )**  
turns on traces for nodes in *l\_nodes*. (Alternate syntax: t node1 node2 ...)

**( ut 'l\_nodes )**  
turns off traces for nodes in *l\_nodes*. (Alternate syntax: ut node1 node2 ...)

**( invoc '( vec\_name g\_val1 g\_val2 ... ) )**  
checks the type of *vec\_name*. If the vector is one of the numeric types (hex, dec, oct, or bin) then it assigns the numeric value *g\_val1* to the vector. Otherwise, it treats the vector as a bit vector and assigns the value *g\_val1* to the first node in the vector, *g\_val2* to the second, etc.

**( bitinvoc '( vec-name g\_val1 g\_val2 ... ) )**

is like *invec* but forces the input to be in the bit vector form. This function allows elements of the vector to be set to *x* or *u*.

( *openplot* '*file\_name*' )

opens a plot file to receive notifications of reported transitions. The resulting plot file can be processed by the program *mup* to produce plots that resemble logic analyzer displays.

( *closeplot* '*l\_arg*' )

closes the plot file. The argument is ignored.

( *markplot* *marker* )

inserts a marker with the name *marker* in the plot file.

( *s* '*l\_arg*' )

runs a simulation step for *incr* simulated time and generates a report at the end. (See *def-report*.)

( *c* '*i\_arg*' )

runs a two-phase non-overlapping clock for *i\_arg* cycles. This assumes that the clock nodes are called *phi1* and *phi2*. Each of the 4 periods is *incr* long. At the end of *i\_arg* cycles a report is attempted using the user's declared format.

( *vecnames* *arg* )

*arg* should be either vector definition or a symbol with a vector definition. It prints out the names and current values of the vector.

( *vecnodes* *arg* )

*arg* should be either vector definition or a symbol with a vector definition. It returns the list of component nodes of the vector.

( *unchanged-since* '*n\_time*' )

returns a list of nodes that have not changed since *n\_time*. This is useful for helping to decide whether your simulation has adequate coverage.

( *unchanged* '*l\_arg*' )

is shorthand for (*unchanged-since* 0). The argument is ignored.

( *chflag* '*l\_arg*' )

sets the "STOPONCHANGE" flag for the nodes in *l\_arg*.

( *unchflag* '*l\_arg*' )

clears the "STOPONCHANGE" flag for the nodes in *l\_arg*.

### 7.3. Functions Intended to be Called by Other Functions.

There are a lot of these functions. You are invited to look at *uwsim.l*.

In addition, they provide examples of how many of the elements of this language are utilized.

# **NETLIST & RNL**

## **Tutorial for Beginners**

-----  
*Rudolf W. Nottrott*

**UW/NW VLSI Consortium  
Sieg Hall, FR-35,  
University of Washington,  
Seattle, WA 98195**

This document is intended as a guide for people who want to acquire a basic working knowledge of the RNL digital circuit simulator and the NETLIST network description program in the UW/VLSI VAX 11/780 environment. Examples are given for the preparation of logic network description files and the production of the corresponding *.sim* and binary files for input to RNL. Next, instruction on the use of RNL commands to set up a clearly defined network state for a simulation is provided. Performing actual simulations of some of the networks defined previously, frequently needed commands, such as those for setting circuit values, asking about node information, running a simulation step, etc., are explained and applied to the network in both interactive and batch mode. Further references are listed in the appendix.

If you are not familiar with the UNDX operating system, read the introductory document UNDX-quick.

## **Table of Contents**

1. **Format Conventions for this Document**
2. **Overview of the Position of NETLIST and RNL in the UW/VLSI Tool Environment**
3. **Logic Network Descriptions for NETLIST**
  - 3.1 **A CMOS Inverter as a Simple Example**
    - 3.1.1 **The NETLIST Logic Circuit Description of an Inverter**
    - 3.1.2 **Processing the Description File with NETLIST and PRESIM**
  - 3.2 **NETLIST Description File for a Ten-Bit Shift Register (Made from Latches and MS-Flip-Flops)**
    - 3.2.1 **Defining Macros as Building Blocks (Latches and Flip-Flops) - Building Block Library**
    - 3.2.2 **Making the Register with Macros from the Library - Loops and Indexed Symbols**
    - 3.2.3 **Processing the Register Description File with NETLIST and PRESIM**
    - 3.2.4 **Converting a Network Description into a Macro**
    - 3.2.5 **Sizing of NETLIST Functions with Two or More Transistors (CINVERT, CLKINV, etc.)**
4. **Circuit Simulation with RNL**

- 4.1 Interactive Command Input and Batch Command Input
- 4.2 Practicing RNL Simulations - The Shift Register
- 4.3 Printing RNL Output Using MTP
- 4.4 Displaying RNL Output on a Graphics Terminal or Plotting it on an HP 7220 Plotter

5. Summary and Outlook

Appendix 1 - Further References

Appendix 2 - Description of the .sim file of the example "inverter" (section 3.1.2)

Appendix 3 - Preparation of a simple "config" File

Appendix 4 - The "alias" file of the example "shift" (section 3.2.3).

Appendix 5 - Notice to Users of this Document

**1. Format Conventions for this Document:**

The UNIX system prompt is given as a small percent sign ( % ) in bold-face type.

(The RNL command interpreter has no prompting sign, which can sometimes be confusing.)

All information RNL prints out in response to your entries, including error messages, are indicated in small bold-face type. If a short description of what RNL returns to you is given instead of the actual RNL response, it is also shown in small bold-face type.

Your entries are indicated in bold-face type, normal size.

Certain commands and phrases within the text are also printed in bold-faced letters for emphasis.

Program names within the text are frequently capitalized for emphasis (e.g. PRESIM for presim).

File names are indicated in *italic type* (unless they are part of an entry sequence, in which case they are shown in bold-face as is everything else that is part of an entry).

<CR> stands for the RETURN key.

^ (Caret) stands for the CONTROL key (frequently labeled CTRL). If ^ precedes a character, the CONTROL key has to be pressed and held down while the character key is pressed.

DELETE and <DEL> stand for the DELETE key.

## 2. Overview of the Position of NETLIST and RNL in the UW/VLSI Tool Environment

The *sim* file plays a central role in connecting NETLIST and RNL with the rest of the UW/VLSI tools, as well as with each other (see Figure 1). The name *sim* file derives from the mandatory file name extension *sim*.

In this tutorial, we will use NETLIST to produce a *sim* file from a logic network description file, then transform the *sim* file with the program PRESIM into a binary network description file for input to RNL. We will not deal with the generation of the *sim* file with other tools, such as MEXTRA, nor will we consider the possibility of using the *sim* file for input to programs other than RNL.

RNL must be given two sets of information for a simulation run. These are:

- (a) a description of the network to be simulated and
- (b) the commands to control the simulation run.

The network description specifies the elements of the network (such as transistors, NANDs, etc.) and the way they are interconnected. The RNL command input performs functions such as setting the initial state of the network, providing signals to be applied to the network (input signals, clock, ...), specifying the format of the simulation report, etc.

We will deal with the network description first, and later use the networks defined in this way to illustrate the generation of the RNL command input.

The RNL command input can be entered in two ways: interactively, or via a command file submitted to RNL (batch mode). You may use a command file to initialize the network or to apply complex stimulus signals, and then continue to work with RNL interactively. In interactive mode, you can do simple simulations and develop programs in RNL LISP, which may be run in batch mode subsequently. In most RNL sessions you will probably find yourself alternating between interactive and batch mode.

There are conventions for the names of the files to be processed by NETLIST, PRESIM and RNL. These conventions will be observed in this tutorial. All files related to a particular circuit are given the same main name followed by a period (.) and an extension. The binary input file for RNL is an exception to this rule - it has no extension.

The meanings of the extensions are:

<i>.net</i>	network description file, input to NETLIST
<i>.sim</i>	intermediate file, output of NETLIST and input to PRESIM; this file is a true "mediator" - it forms the connection not only between NETLIST and RNL, but also between NETLIST and the other VLSI tools, and RNL and the other VLSI tools.
<i>.l</i>	command file for RNL (batch mode)
<i>.al</i>	alias file produced by NETLIST*

\* Normally you do not have much to do with the *.al* file, it is created automatically by NETLIST and used automatically by PRESIM.

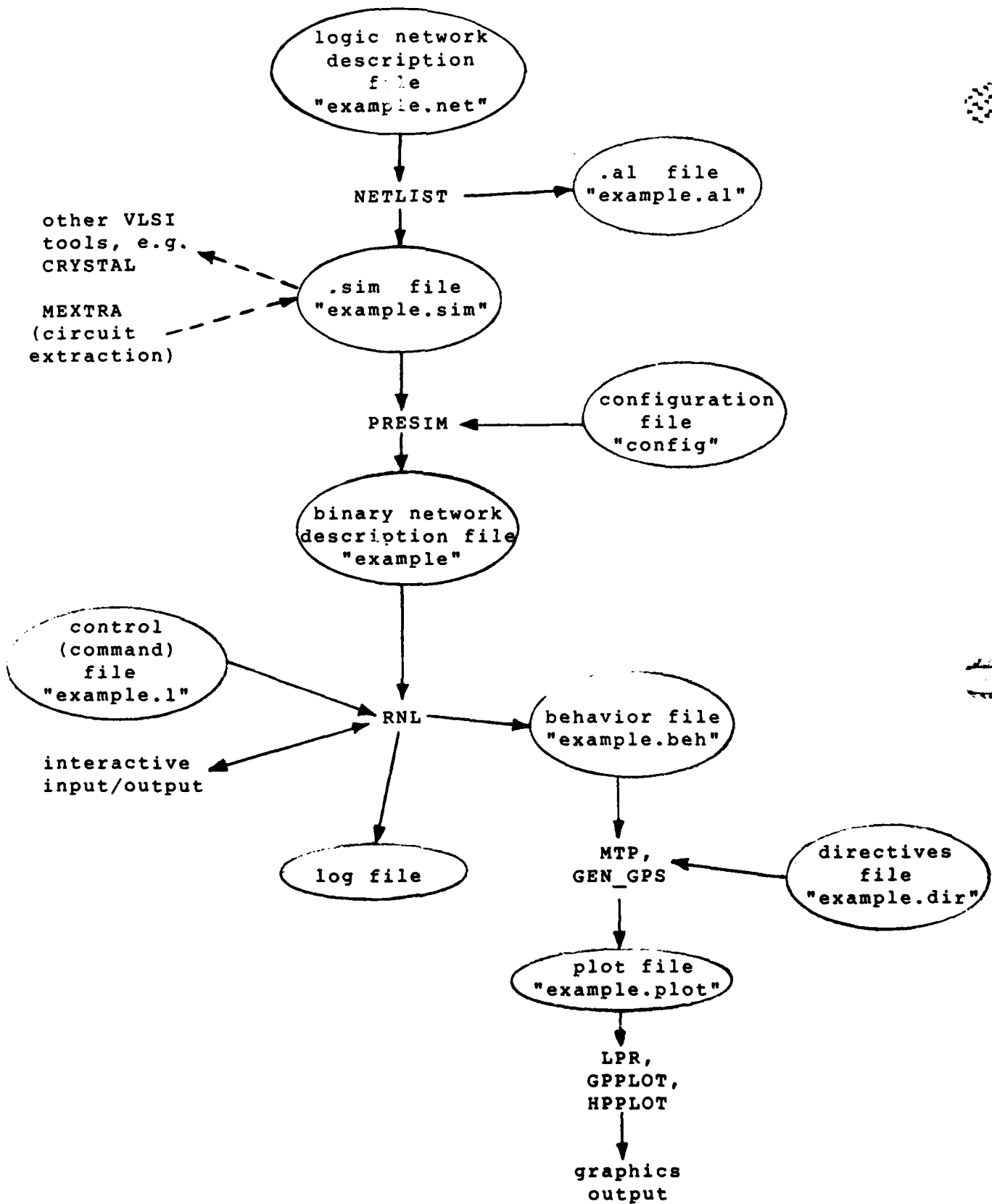


Figure 1 Position of NETLIST and RNL in the UW/VLSI Tool Environment



AD-A146 444

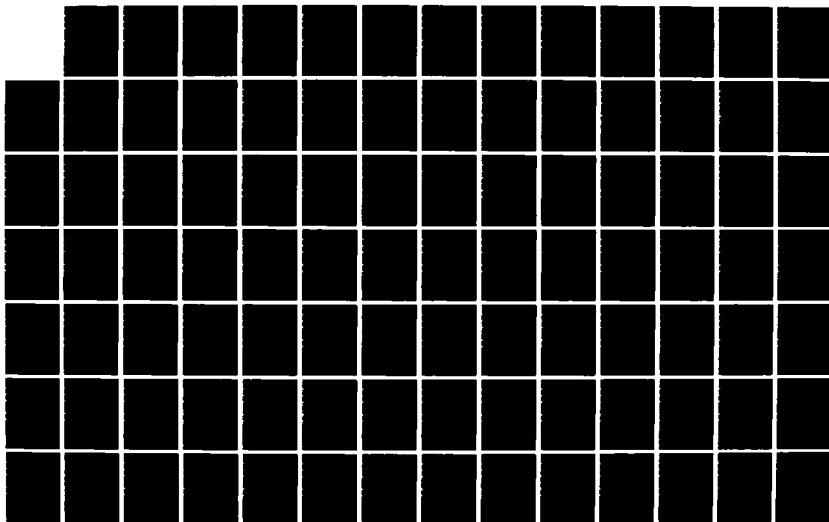
VLSI DESIGN TOOLS REFERENCE MANUAL RELEASE 20(U)  
WASHINGTON UNIV SEATTLE DEPT OF COMPUTER SCIENCE  
AUG 84 TR-84-08-07 MDA903-82-C-0424

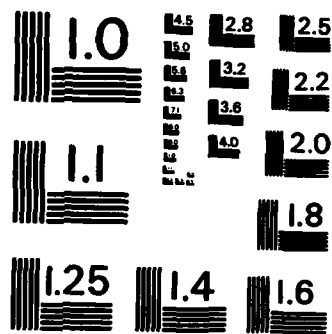
3/4

UNCLASSIFIED

F/G 9/5

NL





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS - 1963-A

no extension: binary network description file, output of PRESIM, input to RNL

You need not follow these file name conventions, but it is strongly recommended since it makes the communication between various users much easier.

### 3. Logic Network Descriptions for NETLIST

#### 3.1. A CMOS Inverter as a Simple Example

##### 3.1.1. The NETLIST Logic Circuit Description of an Inverter

Recall from section 1. that RNL needs to know the network before you can start your simulation\*. We will describe here how the network can be specified for NETLIST in a logic network description file using a LISP-like command syntax.

You therefore should familiarize yourself with the basics of LISP, if necessary. To help you with this, this section will provide an example of a logic network description and an implicit description of some important LISP properties, but will omit much of the detail and the intricacies of LISP.

Figure 2. shows the circuit diagram of our first example, a simple CMOS inverter. We will prepare the logic network description for the inverter according to this diagram.

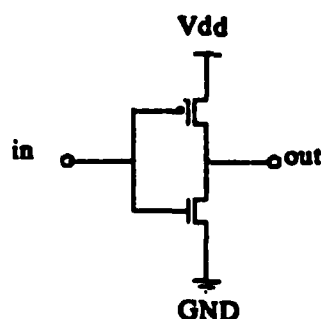


Figure 2 CMOS Inverter

\* However, it is possible to add to your network in the course of the simulation.

The general form of a NETLIST (and RNL) command is:

```
(command_name argument_1 argument_2 argument_3 ...)
```

In most cases the parentheses are required. You will find later in this tutorial examples for cases where the parentheses can be left out. (In particular, RNL has a syntax simplification designed to save you some typing of parentheses).

Here is a listing of the logic network description, followed by the explanation of its commands (write the network description into a file named *inverter.net*):

```
; (1) NETWORK DESCRIPTION FOR A CMOS INVERTER
; -----

; (2) DECLARATION OF THE NODES IN THE NETWORK
(node in out)

; (3) P-CHANNEL ENHANCEMENT TRANSISTOR (PULL-UP)
(ptrans in out Vdd 8 8)

; (4) N-CHANNEL ENHANCEMENT TRANSISTOR (PULL-DOWN)
(etrans in GND out 4 8)

; (5) SPECIFYING AN INTERCONNECT CAPACITANCE FOR THE OUTPUT NODE
(capacitance out 0.83)
```

Hereafter, the numerals enclosed in parentheses will be used to indicate each part of the description file.

- (1) Note that a semicolon causes the rest of the line to be treated as a comment, i.e., to be ignored by the NETLIST program. Blank lines are also ignored\*.

This first comment serves as a title to the network description file.

- (2) You must declare any node you want to name for subsequent reference. (you could think of this declaration as bringing the nodes named by you to the attention of RNL)\*\*. There are a few exceptions to this rule, however. Some nodes, common to most circuits, are known to RNL without declaration. These are the ground and drain voltage potential connections\*\*\* symbolized GND and Vdd (the symbols GND and Vdd are not sensitive to upper or lower case, so

\* You should make ample use of such comments and blank lines to make your network file as "readable" as possible.

\*\* There are almost always additional nodes named by NETLIST (or other programs producing a .sim file). This happens, for example, when NETLIST processes a macro which has internal (local) nodes. Such names go undeclared. You will find many of them in most .sim files.

\*\*\* We use the term "drain voltage potential" despite the fact that in many cases the drain of a transistor may be connected, not to Vdd, but to any other node. Notably this is the case with transmission gates.

`gnd` and `vdd` are equivalent symbols).

Nodes are declared with the command

`(node n1 n2 n3 n4 ...),`

where `n1`, `n2`, `n3`, `n4`, .. are the names of the nodes to be referred to in the network.

- (3) The declaration of the nodes has provided the "skeleton" for the network. Next you must "fill in" the remainder of the circuit. For a transistor this is done with a command of the form
- `(transistor-type gate source drain width length).`

**Transistor-type** represents a mnemonic for various types of transistors, such as

`ptrans` for p-channel enhancement-mode transistor

`etrans` for n-channel enhancement-mode transistor

`dtrans` for n-channel depletion-mode transistor

(see NETLIST User's Guide for more available transistor types and other circuit elements.)

**Gate, source, and drain** represent the names of the nodes to which the gate, the source, and the drain of the transistor are connected.

The pull-up of the inverter is specified as a p-channel enhancement-mode transistor, and the appropriate nodes are "in", "out", and "Vdd".

The width and the length of the transistor's gate area in units of lambda may be specified optionally. If omitted, both width and length default to 2 lambda. Our pull-up is given a width of 8 lambda and a length of 8 lambda.

The width and the length of the gate area determine the resistance of the transistor\*. In this way you can influence the ratio of the pull-up to pull-down.

- (4) The pull-down is specified, analogously to (3), as an n-channel enhancement-mode transistor with a gate width of 4 and a gate length of 8.
- (5) The final element to be specified in the inverter is the interconnect capacitance. The command

`(capacitance out 0.03)`

tells NETLIST that a capacitance of 0.03 pF is to be inserted between the nodes "out" and GND. (One side of a capacitance specified with this command is always to GND). The specification of this capacitance is an estimate of the load capacitance of the inverter.

\* RNL determines the resistance by looking up a two-dimensional table in which the dimensions are length and width. In this way the influence of the geometry of the gate area on the effective (empirical) resistance may be taken into account.

### 3.1.2. Processing the Description File with NETLIST and PRESIM

After the logic network description has been written to the file *inverter.net*, it has to be processed with the NETLIST and PRESIM programs.

Substitute "inverter" for "example" in the file names from Figure 1. (You already prepared the file *inverter.net*.)

Then enter:

```
% netlist inverter.net inverter.sim <CR>
```

This causes NETLIST to process the network description file *inverter.net*, writing its output to the file *inverter.sim*. If you omit the filename *inverter.sim*, NETLIST will display the output on the screen<sup>\*\*</sup>. (Enter % man netlist or see NETLIST User's Guide to get more information about optional parameters for NETLIST).

If everything worked out correctly, the only response you will get is the UNIX prompting sign (%).

You may want to look at the *inverter.sim* file produced by NETLIST. Its content is listed and analyzed for this example in the appendix.

The next step is to process *inverter.sim* with PRESIM. PRESIM transforms the transistors in the *sim* file into resistors of equivalent size. This is done because RNL uses resistor models for the transistors and estimates transition time delays from the equivalent network formed by the resistors and the circuit capacitances.

There is an optional configuration file which can be used to give to PRESIM technology-dependent parameters, such as the specific resistances of the transistor channels. If you do not use this configuration file in the PRESIM run, default values for the specific channel resistances are assumed. The assigned default values are the same for all the different transistor types, which results in a resistance ratio of 1 if the gate areas for the pull-down and the pull-up transistors are sized equally.

An explanation is given in the appendix on how to prepare a simple configuration file to change the channel resistance of the p-channel transistor to twice the value of that of the n-channel transistor. Read this appendix or simply prepare the very short *config* file (six lines) from the listing there. You can then run PRESIM with the *config* file as a parameter:

```
% presim inverter.sim inverter config <CR>
```

This will cause PRESIM to process *inverter.sim*, putting the output into the file *inverter*. This output is a binary file.

PRESIM will give you some information about what it did:

```
Version 4.3
```

```
8 nodes, transistors: nch=1 intrinsic=0 p-chm=1 dep=0 low-power=0 pullup=0 resistor=0
```

```
Total transistors eliminated = 2
```

<sup>\*\*</sup> In fact, it will go to the UNIX standard output, which is normally assigned to the screen. Of course, you may redirect the standard output (e.g. to a file).

First, it tells you its version number, which is 4.2. Then, you are informed that eight nodes were found in the network. If you look at Figure 2, you might count only four nodes, but PRESIM counts some of the nodes more than once.

PRESIM tells you how many transistors of the various types it found in the circuit and how many transistors it "eliminated".

You could now use the *inverter* file as the binary network description for RNL and run a simple simulation. However, the example of the inverter was only intended to be a simple exercise to give you a feeling for the way networks are described using NETLIST. We will consider the network description of a more complex network before proceeding to an actual simulation.

### 3.2. NETLIST Description File for a Ten-Bit Shift Register Made from Latches and Flip-Flops

#### 3.2.1. Defining Macros as Building Blocks (Latches and Flip-Flops) - Building Block Library

We are going to build the shift register in a modular fashion as illustrated in Figure 3a through 3c. First we make a latch from inverters (3a), then we put together two latches to get a master-slave flip flop (3b), and finally we chain ten flip-flops to build the shift register (3c).

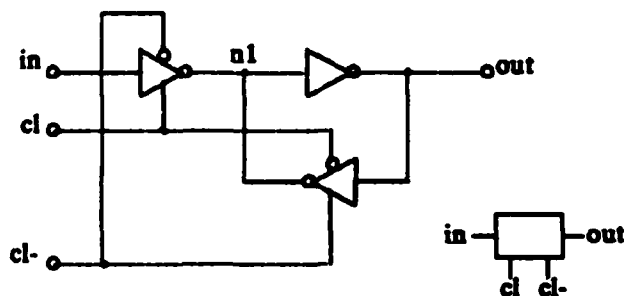


Figure 3a Latch

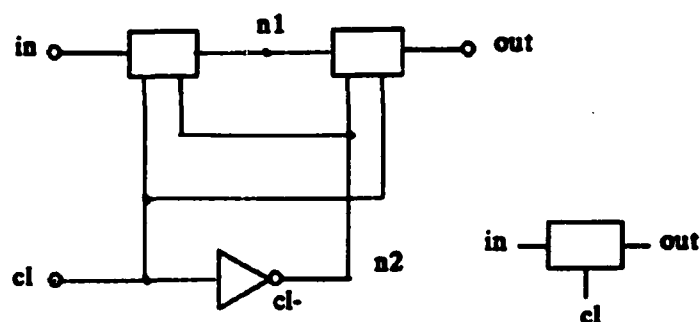


Figure 3b MS Flip Flop (made from Latches)

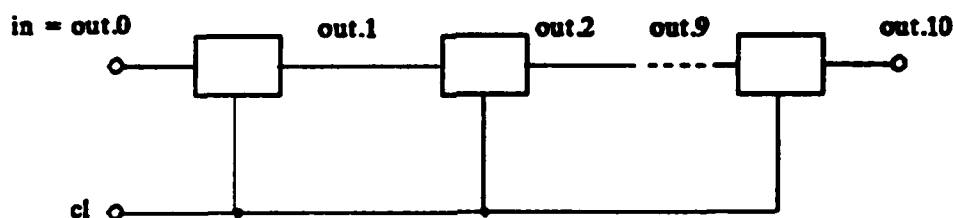


Figure 3c Shift Register (made from ms-ff)

Since it can already be seen that we may need latches and flip-flops in future designs, we will define these building blocks such that we can later call them without having to redefine them. This is done with macro definitions, which can be stored in a library file and easily loaded into future network description files. The library file, which we shall call network library, has the same format as the *.net* files. We give it the name *lib.net*.\*

Here is the listing of the macro definition of the latch, followed by the explanation of its statements (see also Figure 3.a):

**; (1) MACRO DEFINITION FOR A CMOS LATCH**

**; -----**

\* It is not necessary to store the macro definitions in a library file. You may choose to write the macro definitions directly at the beginning of the network description file.



**; (2) NAMING THE MACRO AND ITS PARAMETERS****(macro latch (out in cl cl-))****; (3) DECLARATION OF THE NODES LOCAL TO THE LATCH****(local n1)****; (4) FIRST CLOCKED CMOS INVERTER****(clkinv n1 in cl cl-)****; (5) UNLOCKED CMOS INVERTER****(clavert out n1)****; (6) SECOND CLOCKED CMOS INVERTER****(clkinv n1 out cl- cl)****; (7) CLOSING PARENTHESIS FOR THE MACRO****)**

- (1) This is the title identifying the library entry.

- (2) A macro definition has the general format

```
(macro name (param1 param2 param3 ...)
  body of the macro
)
```

Note the closing parenthesis after "body of the macro", and make sure that you never forget them in any macro definition. You should invent short and descriptive names, but do not use the name of any of the other NETLIST functions (as listed in the NETLIST User's guide.). The body of the macro is made up of the statements (2) through (6).

We give our macro the name "latch". The name is followed by a list of parameters param1, param2, param3, ... These parameters represent the values to be used when the macro is called later. In the latch macro, they represent the names of the nodes that are used to connect the latch to other circuits.

- (3) There is one node to which one need not refer when the latch is used later. This node, "n1", is only of local importance to the latch. Therefore it is declared as a local node in the latch macro. Locally declared node names may be declared and re-used in other macros, since they are considered free symbols outside the macro of their declaration.
- (4) As in the previous example of the inverter, the nodes form only a "skeleton" for the network which must be "filled in" with the circuit elements. The circuit elements are two clocked inverters and an unlocked inverter. These elements are available in standard form as commands for the network description. (This means we need not have taken the trouble to describe the inverter with single transistors in section 3.1. However, we did this as an example of a simple circuit, and to demonstrate the use of transistors in a network description.)

A clocked CMOS inverter is specified with a command of the form  
(clkinv out in clk clk-).

"clkinv" is a mnemonic for "CMOS clocked inverter". "out", "in", "clk", and "clk-" represent the names of the nodes to which the output, input, clock input, and negated clock input are connected (the clock input is the base of the n-channel transistor, the negated clock input is the base of the p-channel transistor). In (4), a clocked inverter is connected to the respective nodes out, in, cl, and cl-. (The gate sizes and ratio of the clocked inverter "clkinv", and the devices "cnand", "cnor" and "cinvert", can be changed with "width" and "length" values and with the "ratio" command; see section 3.2.5 and NETLIST User's Guide.)

- (5) A simple unclocked inverter is inserted. The general command for the specification of this CMOS inverter is (cinvert out in), with out and in representing the names of the nodes to which the output and the input of the inverter are connected.
- (6) The second clocked CMOS inverter is specified analogously to (4).

After completing the macro definition, save it in the library file *lib.net*.

In the following definition of the master-slave flip-flop (see also Figure 3b), you can simply call the latch by its macro name.

The macro definition of the master-slave flip-flop is analogous to that of the latch. Its statements, followed by the explanation of their meanings, are listed below:

**; (1) MACRO DEFINITION FOR A CMOS MASTER-SLAVE FLIP-FLOP**

**;** -----

**; (2) NAMING THE MACRO AND ITS PARAMETERS**

(macro maff (out in cl)

**; (3) DECLARATION OF THE NODES LOCAL TO THE FLIP-FLOP**

(local n1 n2)

**; (4) FIRST LATCH**

(latch n1 in cl n2)

**; (5) SECOND LATCH**

(latch out n1 n2 cl)

**; (6) CMOS INVERTER**

(cinvert n2 cl)

**; (7) CLOSING PARENTHESIS FOR THE MACRO**

)

Most of the statements in this macro definition will already be familiar to you. Note that the previously defined macro "latch" is used in the same way as other circuit elements. If you had not defined the latch yourself, you might not even know whether "latch" is a macro or a basic NETLIST function. This property allows you to define successively more complex building blocks and nevertheless use them with the same ease as the "primitive" functions.

As we did with the macro definition for the latch, we now add the macro definition for the master-slave flip-flop to our library file *lib.net*. It must be inserted after the latch, since it uses "latch" as a circuit element and will therefore call the "latch" macro.

### 3.2.2. Making the Register with Macros from the Library - Loops and Indexed Symbols

Looking at Figure 3c, it is now easy to make the ten-bit (or any number of bits) shift register by chaining ten of the flip-flops defined in our macro library *lib.net*. We could write down the call for "msff" with the appropriate parameters ten times, but NETLIST has the facility of a loop and indexed symbols, which makes the specification of such repetitive elements as the flip-flops of the shift register very compact. (In accordance with the file name conventions, write the following network description into a file named *shift.net*.):

```

; (1) CIRCUIT DESCRIPTION FOR THE 10-BIT SHIFT REGISTER
; -----

; (2) LOADING THE FUNCTIONS FROM THE MACRO LIBRARY
(load "lib.net")

; (3) NODE DECLARATION FOR THE NETWORK
(node in out cl)

; (4) LOOP CALLING THE MASTER-SLAVE FLIP-FLOP 10 TIMES
(repeat 1 1 10
  (msff out.1 out.(- 1 1) cl)
)

; (5) ASSIGNING AN ADDITIONAL NAME TO THE INDEXED NODE OUT.0
(connect in out.0)

```

- (1) This is the usual title.
- (2) Load the macro library *lib.net*, which has the effect of inserting the macro definitions for latch and msff before the description of the shift register.

- (3) Declare the network nodes to which you want to refer. Remember, there are two kinds of node declarations: global, as here and in the example of the inverter (section 3.1); and local, as in our macro definitions.

- (4) This part illustrates two new facilities which we have at our disposal when we want to specify the description of network structures that contain the same sub-circuit repetitively.

In most cases of such network structures, as in our shift register, the regular pattern of sub-circuits makes it possible to refer to their nodes with a collective name followed by an index.

out.0, out.1, out.2, ..., out.10 (see Figure 3c) are examples for indexed node names. "out" is the collective node name for a group of nodes having a similar function or position in the network; ".0", ".1", ... are the indices uniquely identifying each of the individual nodes.

Note one other application of indexed nodes in the "node" command of (3). Indexed nodes can be declared by simply declaring their collective node name. It is not necessary to list each individual node. Thus, our declaration of "out" in (3) represents out.0, out.1, out.2, ... out.10.

Generally, an index can be represented by any symbol or expression. In the "repeat" loop, it is given as "i" and "(- i 1)", respectively. (- i 1) is the LISP form of subtracting 1 from i, and just one example of how an index can be calculated from an expression.

The symbolic index enables us to use a loop calling our master-slave flip-flop ten times. A loop has the format

```
(repeat loop-index start-value end-value
      body of the loop
)
```

The "body of the loop" can be any sequence of commands. The loop in (4) starts with the index "i" set to 1. "i" is increased by 1 after each call to the flip-flop. In this way the loop specifies flip-flops with connections to

```
out.1, out.0, cl
out.2, out.1, cl
out.3, out.2, cl
.
.
.
out.10, out.9, cl
```

After "i" has reached the value 10, the loop is exited.\*

\* Upon leaving the loop, the value of the symbol of the loop index will be restored to the value it had before entering the "repeat" form.

- (5) The network description using "repeat" and indexed nodes looks very compact, but a few nodes were designated cumbersome names. Rather than using the name "out.0" for the input to the shift register (see Figure 3c), we will call it "in". This can easily be accomplished with the `connect` command, which equates the names "in" and "out.0". Another application for the `connect` command is in connecting two or more nodes electrically.

### 3.2.3. Processing the Shift Register Description File with NETLIST and PRESIM

You should be familiar with the procedure for processing a description file with NETLIST and PRESIM, from section 3.2.1. However, be aware of one additional file created by NETLIST. This file is called an alias file and it contains, for each node, all the different symbolic names that have been assigned to that node. Remember that you specified explicitly with the `(connect in out.0)` statement that the names "in" and "out.0" were to denote the same node.

The alias file created by NETLIST has the same main name as the other files related to the network, which is *shift* for the shift register, followed by the extension *.al*. A short explanation of the *shift.al* file is given in the appendix.

Substitute "shift" for "example" in the file names from Figure 1. (You have already prepared the file *shift.net*.)

Then enter:

```
% netlist shift.net shift.sim <CR>
```

This will cause NETLIST to process the network description file *shift.net* in the same way as explained before for the inverter.

(Feel free to inspect the *shift.sim* file produced by NETLIST, which should not be difficult since you know how interpret *.sim* files from the example *inverter.sim* analyzed in the appendix.)

To process the description file with PRESIM enter:

```
% presim shift.sim shift config <CR>
```

PRESIM gives you the following information:

```
Version 4.2
```

```
139 nodes; transistors: cmh=110 intrinsic=0 p-chm=110 dep=0 low-power=0 pullup=0 resistor=0
```

```
Total transistors eliminated = 228
```

It is very much similar to the case of the inverter except that the shift register circuit is much bigger.

You have now seen, for several examples, how to produce the network description needed by RNL. The next section examines a special aspect of network descriptions (converting network descriptions to macros). You may want to skip this now, and proceed to actual simulations of two of the networks we have described so far.

### 3.2.4. Converting a Network Description into a Macro

You have seen how to use macros as building blocks in the description of a network. A network description can easily be turned into a macro in the following way:

- Specify as parameters the nodes you want to access when you call the macro. In the example below, (output in ci) are the parameters for the output, input and clock connections. Later, before you call the macro, you must "globally" declare (with the "nodes" command) the actual names for the respective nodes.
- Declare the remaining nodes as local nodes. Be careful not to use names that are to be used as global node names in the "main" network. In the example below, the indexed name "o" is used to represent all the individual indexed nodes o.0, o.1, o.2, ..., o.10 (formerly out.0, out.1, out.2, ..., out.10).

Here is the listing of a possible macro definition for the ten-bit shift register, derived from the shift register's network description:

```

; (1) MACRO 10-BIT SHIFT REGISTER
-----
;  THIS MACRO MAY BE CALLED ONLY WHEN THE "LATCH" AND
;  "MSFF" MACROS HAVE BEEN DEFINED OR LOADED
;  PREVIOUSLY.

; (2) MACRO DEFINITION
(macro shiftreg_10 (output in ci)

; (3) NODE DECLARATION FOR THE NETWORK
(local o)

; (4) LOOP CALLING THE MASTER-SLAVE FLIP-FLOP 10 TIMES
(repeat 1 1 10
  (msff o.i o.(- 1 1) ci)
)

; (5) ASSIGNING AN ADDITIONAL NAME TO NODES o.0
;  AND o.10
(connect in o.0)
(connect output o.10)

; CLOSING PARENTHESIS FOR THE MACRO.
)

```

After defining the macro, you can add it to the library *lib.net* and use it in the same manner as you have used "latch" and "msff".

### 3.2.5. Sizing of NETLIST Functions with Two or More Transistors (CINVERT, CLKINV, etc.)

In section 3.2.1 the NETLIST functions "cinvert" and "clkinv" were used to define the macros "latch" and "msff". We did not specify any gate sizes there, so the default values were assumed. You will see here what the default values are and how they can be changed.

In NMOS functions, such as inverters, NANDs, NORs, etc. (represented by the NETLIST functions "invert", "nand", "nor")\* each individual transistor can be identified by the node to which its gate is connected. This is either one of the input nodes, or, in the case of the depletion-mode pull-up transistor, the output node (since the gate of a depletion-mode pull-up is connected to its source, which is the output). You can specify the gate size for each transistor by specifying width and length together with the node to which the gate is connected in the following manner:

```
(invert (out width-o length-o) (in width-i length-i))
(nand (out width-o length-o) (in1 width-1 length-1) (in2 width-2 length-2) ...).
```

Thus, (invert (out 4 6) (in 8 10)) creates an NMOS inverter whose enhancement-mode pull-down has a gate area of 8 by 10 lambda, and whose depletion-mode pull-up has a gate area of 4 by 6 lambda. The case is similar for the "nand" and the "nor", the only difference is that you have more than one input. The default gate sizes are 2 by 2 lambda for an enhancement-mode transistors and 2 (width) by 8 (length) lambda for a depletion-mode transistor.

In CMOS devices the situation is different. Normally an input connects to two gates - one gate of a p-channel transistor and one gate of an n-channel transistor. This makes the sizing specifications somewhat more complicated, since a node does not any longer uniquely identify an individual transistor. NETLIST permits you to specify width and length together with a node as in the NMOS case, e.g.,

```
(cinvert out (in width length))
```

However, these values determine only the gate width and length of the n-channel transistor and the gate length of the p-channel transistor. Defaults are 2 lambda. You can set the width of the p-channel transistor to a multiple of the width of the n-channel transistor with the command

```
(ratio value)
```

which must precede the function it is to affect. For example,

```
(ratio 3)
(cinvert out (in 4 6))
```

sets the "ratio" for "cinvert" (and all following CMOS functions until the next "ratio" command is encountered) to 3. The n-channel transistor (pull-down) of the inverter gets a gate area which is 4 lambda wide and 6 lambda long. The p-channel transistor (pull-up) gets a gate area which is  $3 * 4 = 12$  lambda wide and 6 lambda long.

The default "ratio" is 2.

If a node connects to only one gate, width and length of the gate area are set in the same manner as

\* We did not use them but concentrated on CMOS functions. If you want to know more about them, see the NETLIST User's Guide.

described above for NMOS. Thus, in the case of a clocked inverter, you can specify the dimensions of the gate area of the two transistors connected to the input node in the same manner as in the case of "cinvert", whereas the dimensions of the two transistors whose gates are connected to "cl" and "cl-" are specified independently:

```
(ratio 3)
(clkinv out (in 4 6) (cl 8 10) (cl- 12 14))
```

The gate dimensions of the two transistors forming the inverter are the same as in the "cinvert" above. The gate area of the p-channel gating transistor connected to "cl-" is 12 lambda wide and 14 lambda long, the gate area of the n-channel gating transistor connected to "cl" is 8 lambda wide and 10 lambda long.

The sizes of the other CMOS functions available in NETLIST are set in a similar manner (see NETLIST User's Guide).

#### 4. Circuit Simulation with RNL

##### 4.1. Interactive Command Input and Batch Command Input

You have now completed all the necessary preparations to run a simulation of one of the circuits described previously. We will run RNL and start out with our simple inverter.

```
% rnl <CR>
```

RNL comes up with its version number

```
Version 4.2
```

and waits at the beginning of the next line for your command input. (RNL does not have prompting sign.) Every command you enter now is immediately executed and, if necessary, commented by RNL. This is why this mode of operation is called the interactive mode. Correspondingly, this way of entering commands is called interactive command input.\*

Before starting on your simulation, you must load the two files *uwstd.l* and *uwsim.l*, containing function definitions for RNL.

```
(load "uwstd.l") <CR>
```

---

\* Just in case you want to exit RNL, the command to leave RNL in an orderly fashion is (exit) or simply exit.



**done**

**(load "uwsim.l") <CR>**

**Loading uwsim.l**

**Done loading uwsim.l**

(The loading of the *uwssd.l* file is implied).

Next, load the binary network description file *inverter*.

**(read-network "inverter") <CR>**

RNL will prompt with information about the network:

**; 8 nodes; transistors: umh=0 intrinsic=0 p-chan=0 dep=0 low-power=0 pulldn=0 resistor=0**

There is a simple command ("s" - we will use it shortly) to run a simulation step for an amount of time defaulted to 100 ns. To change this, a variable *incr* can be set. *incr* \* 0.1 ns is the new length of the of the simulation step. For example, an *incr* of 10 results in a simulation step length of 1 ns. The command to assign a value to a symbolic variable\* is (**setq symbol value**).

Let us carry out the simulation in steps of one nanosecond. Set *incr* to 10 so that the product of "incr" and the internal step width is 1.

**(setq incr 10) <CR>**

**10**

RNL echoes (returns) the value assigned to *incr*.

Frequently it is convenient to refer to a group of nodes, rather than to one individual node. You can denote a symbolic name for a list of node names with "setq". We will give the name "nodes" to the list of the two nodes "in" and "out".

\* Strictly speaking, "incr" is a LISP symbol. Symbols in LISP are not quite the same as variables in other programming languages. In many cases however, as with "incr", they act just like "conventional" variables. We therefore will frequently designate as variables objects which should strictly be called symbols.

```
(setq nodes '(in out)) <CR>
```

```
(in out)
```

Again, RNL returns the value it assigned to the variable, which is the list (in out).

The final step is to specify details for the reports on the simulation step. There are two standard report forms available in RNL.

The first type of report lists the state of nodes whenever this state changes. We want to obtain such a report for the changes in our "nodes", i.e. "in" and "out". Each node has a "change-flag" telling RNL that such a report is requested. For a list of nodes (n1, n2, ...), this "change-flag" is set with the command (chflag '(n1 n2 ...)). Since we have already defined a list of nodes (in out), named "nodes", we can enter

```
(chflag nodes) <CR>
```

```
:
```

RNL has now set the change-flags of "in" and "out" to "true".

The second type of report lists the state of nodes at the end of a simulation step. To obtain such a report on the nodes "in" and "out", use the "def-report" command:

```
(def-report '("STATE AT END OF SIMULATION STEP" in out)) <CR>
```

```
("STATE AT END OF SIMULATION STEP" in out)
```

The capitalized text in quotes is the title of the report. It is followed by the names of the nodes that are to be included in the report.

Now try a simulation. Setting the input of the inverter to high potential\* is simply done by entering

```
h in <CR>
```

RNL's reply `done` means that it carried out your command. (From now on, we will not mention this "done".)

---

\* We will frequently refer to high potential as High, and to low potential as Low.

"h" is the mnemonic for High, followed by the name of the node to be set to High. (You can specify more names, separated by a blank).

Run a simulation step by entering the mnemonic **s** :

```
s <CR>
```

In accordance with your specifications in "chflag" and "def-report", RNL will reply:

```
Step begins @ 0 ns.
in=1 @ 0
out=0 @ 0.6
STATE AT END OF SIMULATION STEP:
Current time= 1
in=1 out=0
```

After starting up RNL, the starting time for a simulation is always set to zero, so your first simulation step begins at 0. The reports on changes in the states of the nodes "in" and "out" show that "in" was set to High at time zero, and that "out" changed to Low at 0.6 ns. This is exactly what an inverter should do. The time delay in the change of the output is caused by the time needed to load the gate capacitance of the inverter and the time needed to unload the output node capacitance of 0.03 pF (see 3.1.1). The report at the end of the simulation step tells you that the time is now 1 ns, and repeats the state of the nodes "in" and "out", as required in the "def-report" command. (For other commands to run a simulation step see RNL User's Guide).

Now consider another state of the inverter, set the input to Low ("l" is the mnemonic for Low):

```
l in <CR>
```

and do a simulation step:

```
s <CR>
```

```
Step begins @ 1 ns.
in=0 @ 1
out=1 @ 0.6
STATE AT END OF SIMULATION STEP:
Current time= 2
in=0 out=1
```

The report given by RNL is analogous to the one just explained. This time the simulation starts out at 1 ns with the input set to Low. The output changes to High at 0.6 ns (relative to the starting point of the simulation step), again the delay is caused by the gate and output capacitances. At the end of the simulation step the time is 2ns, "in" is Low and "out" is High.

RNL uses three different logic states to characterize the potential of a node. They are:

**High,** symbolized H, with the value 1. Logic high is assumed whenever the simulated voltage level of the node is between a high threshold  $V_{high}$  and 1.

**Low,** symbolized L, with the value 0. Logic low is assumed whenever the simulated voltage level of the node is between 0 and a low threshold  $V_{low}$ .

**Undefined,** symbolized X. An undefined state is assumed whenever the simulated voltage level of the node is between a low threshold  $V_{low}$  and a high threshold  $V_{high}$ .

You can set any node to one of the three states with the following commands:

**h** Set the node to High (shown above).

**l** Set the node to Low (shown above).

**u** Set the node to Undefined.

Setting a node in this way has the effect of connecting the node to a voltage source with zero impedance, thus overriding any other value the circuit might try to impose. Nodes with their values fixed in this way are called input nodes (because they are used like an input to a circuit, and RNL internally puts them on an "input list"). They will stay at the assigned logic level until you release them. You release nodes with the command **x**, followed by the names of the nodes you want to release. After the node has been released, it is free to assume whatever level it wants to assume naturally in the circuit (it will assume this level after an "s" command is executed).

The simple example of the inverter has given you a good idea of how to "operate" RNL in interactive mode: You enter a command and receive an immediate reply. Now exit RNL to prepare a "batch" command file:

```
exit <CR>
%
```

You are back with UNIX.

There is one other mode of RNL operation which we shall call **batch mode**. You can write any sequence of RNL command into a file and specify the name of this file as a parameter when you start RNL. We will call this file "RNL command file", or simply command file. You will almost always want to have such a command file to save yourself the work of keying the "load", "read-network", and other commands that are invariably needed to set up the proper conditions for a simulation.

To see how the command file is used, write the first seven of the above commands into a file *inverter.i*:

```
(load "uwstd.i")
(load "uwaim.i")
```

```
(read-network "inverter")
(setq incr 10)
(setq nodes '(in out))
(chflag nodes)
(def-report '("STATE AT END OF SIMULATION STEP:" in out))
```

Now run RNL again, with the command file *inverter.l* as a parameter:

```
% rnl inverter.l <CR>
```

You will get almost the same replies as before when you entered the commands interactively. (The only difference is that returned values, such as the 10 in (setq incr 10), are not shown.)

Now set the input High and run a simulation step, then set the input Low and run another simulation step. Again, you will get the same answers as before.

In order to set up the proper conditions for a simulation, most commonly one starts out with the execution of the commands from a command file and then continues in interactive mode. RNL LISP programs for time-consuming simulations may be developed in interactive mode, written into a command file, and later run in batch mode.

We will do the simulations of the shift register in this mixed way in the next sections.

#### 4.2. Practicing RNL Simulations - The Shift Register

You can modify on the command file prepared in the previous section and use the modified file when we start up RNL for the simulation of the shift register. Write the following modified commands into the file *shift.l*:

```
(load "uwstd.l")
(load "uwsim.l")
(read-network "shift")
(setq incr 100)
(setq nodes '(in out.10 cl))
(chflag nodes)
(def-report '("STATE AT END OF SIMULATION STEP:" in out.10 cl))
```

Now run RNL again with *shift.l* as the start-up command file:

```
% rnl shift.l <CR>
```

RNL's reply is similar to the one discussed in the previous section.

Let us do some initializing and propagate input through the shift register ( (1) through (17) ).

## (1) Initialize the network

**(sim-init)****52**

When you start up RNL, the state of the nodes is Undefined, i.e. neither High nor Low. This state, symbolized "X", is not very useful at the outset of a simulation. It is preferable to start with a definite, stable state. The **(sim-init)** command tries to help you with this. It returns the number of all Undefined nodes and then sets them to Low. In this case, 52 nodes were set to Low. If the number of nodes set to Low was not 0, you should do a simulation step and propagate the new values through the network (we will do that in (2)). If this leads to Undefined node values again, do another **(sim-init)**.

Repeat the sequence of **(sim-init)** followed by a simulation step until **(sim-init)** returns 0. If you cannot settle the network in this way after four to five repetitions, RNL might not been able to simulate your design properly (for example, if a an input signal and a feed-back signal derived from this input simultaneously drive a node). In such cases refer to section four of the RNL User's Guide.

## (2) Simulation step to propagate the nodes set to Low

**s****Step begins @ 0 ns.****out.10=1 @ 0.2****out.10=0 @ 0.4****STATE AT END OF SIMULATION STEP:****Current time= 10****in=0 out.10=0 ci=0****done**

As a result of propagating the Low nodes from the previous **(sim-init)**, "out.10" changes twice during the simulation step. The reports are analogous to the ones explained in the previous section.

## (3) Next initialization step.

**(sim-init)**

0

This time the number of nodes set to Low by (sim-init) was 0, i.e., the network had been settled to a stable state with the commands in (1) and (2).

- (4) Simulation step to propagate the nodes set to Low.

s

Step begins @ 10 ns.

STATE AT END OF SIMULATION STEP:

Current time= 20

in=0 out.10=0 cl=0

done

As expected, no changes occurred in this step (no nodes were changed in the previous (sim-init)). Reports as usual.

- (5) Give a name to a group of nodes.

```
(setq all_nodes '(in out.1 out.2 out.3 out.4 out.5
                  out.6 out.7 out.8 out.9 out.10))
```

```
(in (-struct- out 1) (-struct- out 2) (-struct- out 3) (-struct- out 4) (-struct- out 5) (-struct- out 6) (-struct- out 7)
(-struct- out 8) (-struct- out 9) (-struct- out 10))
```

We give the name "all\_nodes" to the group of nodes forming the shift path. RNL returns its internal representation of this list of nodes.

- (6) Set the change- flag for the group of nodes.

```
(chflag all_nodes)
```

```
(in (-struct- out 1) (-struct- out 2) (-struct- out 3) (-struct- out 4) (-struct- out 5) (-struct- out 6) (-struct- out 7)
(-struct- out 8) (-struct- out 9) (-struct- out 10))
```

This command sets the change-flag for each node of all\_nodes.

We could have used the command

```
(chflag '(in out.1 out.2 out.3 out.4 out.5
        out.6 out.7 out.8 out.9 out.10)
)
```

to achieve the same result, but instead used the symbol *all\_nodes* to give an example of the usage of a name for a group of nodes.

- (7) Set the clock input to Low.

```
l cl
done

s

Step begins @ 20 ns.
STATE AT END OF SIMULATION STEP:
Current time= 30
in=0 out.10=0 cl=0
done
```

The clock is set to Low, followed by a simulation step. Reports as usual.

- (8) Set all nodes in the shift path to High.

```
(repeat 1 0 10
  (h '(out.(eval i)))
)
10

s

Step begins @ 30 ns.
out.10=1 @ 0
out.9=1 @ 0
out.8=1 @ 0
```



```

out.7=1 @ 0
out.6=1 @ 0
out.5=1 @ 0
out.4=1 @ 0
out.3=1 @ 0
out.2=1 @ 0
out.1=1 @ 0
in=1 @ 0
STATE AT END OF SIMULATION STEP:
Current time= 40
in=1 out.10=1 cl=0

done

```

The repeat command is similar to the one in section 3.2.2, where it was used to make a shift register from ten ms-flip-flops. Of course, the body of the loop is different here. It is the command

```
(h '(out.(eval i))).
```

There are several peculiarities in the form of this command (which you have already seen in its simpler forms, e.g. h out.1).

RNL LISP has a syntax simplification which permits you to write

```
command argument_1 argument_2 argument_3 ...
```

instead of

```
(command '(argument_1 argument_2 argument_3 ...))
```

Therefore, h out.10 is equivalent to (h '(out.10)).

However, the simplified syntax may not be used if the command is part of another command, such as inside the "repeat".

The (eval i) is needed because RNL does not evaluate a symbol if it is preceded by an apostrophe ('). (eval i) returns the value of "i", which varies from 1 to 10.

Since we set the change-flag for "all\_nodes", RNL reports all the changes in the corresponding nodes' values, followed by the usual final report at the end of the simulation step.

## (9) Apply one clock cycle

h cl

done

s

Step begins @ 40 ns.

cl=1 @ 0

STATE AT END OF SIMULATION STEP:

Current time= 50

in=1 out.10=1 cl=1

done

l cl

done

s

Step begins @ 50 ns.

cl=0 @ 0

STATE AT END OF SIMULATION STEP:

Current time= 60

in=1 out.10=1 cl=0

done

We set the clock first to High, then to Low, each time followed by a simulation step. This is equivalent to a clock cycle of the length of two simulation steps.

## (10) Release all nodes in the shift path.

```
(repeat 1 0 10
  (x '(out.(eval t)))
```

)

10

s

Step begins @ 60 ns.

STATE AT END OF SIMULATION STEP:

Current time= 70

in=1 out.10=1 cl=0

done

Recall that we have set all the nodes in the shift path to High (8), which is equivalent to connecting them to Vdd. By setting them to "x", we enable them to assume whatever state they may naturally go to in the course of the simulation. Thus, the "repeat" loop releases all the nodes in the shift path of the register.

- (11) Set input to Low.

1 in

done

s

Step begins @ 70 ns. in=0 @ 0

STATE AT END OF SIMULATION STEP:

Current time= 8

in=0 out.10=1 cl=0

done

With the previous steps, all the cells of the shift register have been set to a state of High. Now we set the input to Low in order to later watch this Low signal shift through the register.

- (12) Shift the Low input for one clock cycle

h cl

done

8

Step begins @ 80 ns.

ci=1 @ 0

STATE AT END OF SIMULATION STEP:

Current time= 90

in=0 out.10=1 ci=1

done

1 ci

done

8

Step begins @ 90 ns.

ci=0 @ 0

out.1=0 @ 0.3

STATE AT END OF SIMULATION STEP:

Current time= 100

in=0 out.10=1 ci=0

done

! out.1 out.2 out.3

out.1=L (vi=0.30 vh=0.00) (0.030400 pf) affects:

input to functions for the following nodes:

15

15

8

8

out.2=H (vi=0.30 vh=0.00) (0.030400 pf) affects:

input to functions for the following nodes:

27

27

20

20

out.3=H (vi=0.30 vh=0.00) (0.030400 pf) affects:

input to functions for the following nodes:

39

39

32

32

done

One clock cycle shifts the Low from the input to the output of the first register cell. At the end of the clock cycle we use the **!** (exclamation sign) command to check the values of the first three cells of the register.

**!** provides the values of the specified nodes, their logic thresholds (normalized to 1), and their capacitances. Next it provides the names of all nodes to which the nodes specified with **!** are inputs. In the example, these node names are numerical. The numerical node names were assigned by NETLIST to nodes inside the shift register. They were not declared in *shift.net* but came with the "latch" and "msff" macros. (If you want to know more about them, you have to scrutinize *shift.sim*.)

(**?** (question mark) is a command similar to **!**. It provides information about transistors for which a node is either gate or source, and about the sum-of-products representation of the node. See RNL User's Guide for more information.)

Using **!** (and **?**), you can "walk" through the network and check node values and connections.

The listing produced by **!** shows that the Low input has moved to "out.1", as it should have after one clock cycle.

- (13) Shift the input a second clock cycle.

b cl

done

s

Step begins @ 100 ns.

cl=1 @ 0

STATE AT END OF SIMULATION STEP:

Current time= 110

in=0 out.10=1 cl=1

done

l cl

done

s

Step begins @ 110 ns.

cl=0 @ 0

out.2=0 @ 0.5

STATE AT END OF SIMULATION STEP:

Current time= 120 in=0 out.10=1 cl=0

done

! out.1 out.2 out.3

out.1=L (vi=0.30 vh=0.80) (0.038400 pf) affects:

input to functions for the following nodes:

15

15

8

8

out.2=L (vi=0.30 vh=0.80) (0.038400 pf) affects:

input to functions for the following nodes:

27

27

28

28

out.3=H (vi=0.30 vh=0.80) (0.038400 pf) affects:

input to functions for the following nodes:

39

39

32

32

done

Reports analogous to (12). The Low input has now moved to "out.2".

- (14) Shift the Low input a third clock cycle.

h cl

done

s

Step begins @ 120 ns.

ci=1 @ 0

STATE AT END OF SIMULATION STEP:

Current time= 130

in=0 out.10=1 ci=1

done

! ci

done

8

Step begins @ 130 ns.

ci=0 @ 0

out.3=0 @ 0.3

STATE AT END OF SIMULATION STEP:

Current time= 140

in=0 out.10=1 ci=0

done

! out.1 out.2 out.3

out.1=L (vi=0.30 vh=0.50) (0.030400 pf) affects:

input to functions for the following nodes:

15

15

8

8

out.2=L (vi=0.30 vh=0.50) (0.030400 pf) affects:

input to functions for the following nodes:

27

27

20

20

out.3=L (vi=0.30 vh=0.50) (0.030400 pf) affects:

input to functions for the following nodes:

39

39

32

32

done

Reports analogous to (12). The Low input has now moved to "out.3".

- (15) Do seven more clock cycles to shift the Low input completely to the end (out.10) of the shift register.

```
(repeat 1 1 7
  (h '(cl))
  (s '(x))
  (l '(cl))
  (s '(x))
)
```

Step begins @ 140 ns. ci=1 @ 0 STATE AT END OF SIMULATION STEP: Current time= 150 in=0 out.10=1  
ci=1

Step begins @ 150 ns. ci=0 @ 0 out.4=0 @ 0.5 STATE AT END OF SIMULATION STEP: Current time= 160  
in=0 out.10=1 ci=0

Step begins @ 160 ns. ci=1 @ 0 STATE AT END OF SIMULATION STEP: Current time= 170 in=0 out.10=1  
ci=1

Step begins @ 170 ns. ci=0 @ 0 out.5=0 @ 0.5 STATE AT END OF SIMULATION STEP: Current time= 180  
in=0 out.10=1 ci=0

Step begins @ 180 ns. ci=1 @ 0 STATE AT END OF SIMULATION STEP: Current time= 190 in=0 out.10=1  
ci=1

Step begins @ 190 ns. ci=0 @ 0 out.6=0 @ 0.5 STATE AT END OF SIMULATION STEP: Current time= 200  
in=0 out.10=1 ci=0

Step begins @ 200 ns. ci=1 @ 0 STATE AT END OF SIMULATION STEP: Current time= 210 in=0 out.10=1  
ci=1

Step begins @ 210 ns. ci=0 @ 0 out.7=0 @ 0.5 STATE AT END OF SIMULATION STEP: Current time= 220  
in=0 out.10=1 ci=0

Step begins @ 220 ns. ci=1 @ 0 STATE AT END OF SIMULATION STEP: Current time= 230 in=0 out.10=1  
ci=1

Step begins @ 230 ns. ci=0 @ 0 out.8=0 @ 0.5 STATE AT END OF SIMULATION STEP: Current time= 240  
in=0 out.10=1 ci=0

Step begins @ 240 ns. ci=1 @ 0 STATE AT END OF SIMULATION STEP: Current time= 250 in=0 out.10=1  
ci=1

Step begins @ 250 ns. ci=0 @ 0 out.9=0 @ 0.5 STATE AT END OF SIMULATION STEP: Current time= 260



```
in-0 out.10-1 cl-0
```

```
Step begins @ 260 ns. cl-1 @ 0 STATE AT END OF SIMULATION STEP: Current time- 270 in-0 out.10-1
cl-1
```

```
Step begins @ 270 ns. cl-0 @ 0 out.10-0 @ 0.6 STATE AT END OF SIMULATION STEP: Current time-
280 in-0 out.10-0 cl-0
```

7

(The simulation report is printed more compactly here to save space. Otherwise it is analogous to the reports in (12) through (14)). At the end of each clock cycle the Low input appears shifted one more cell toward the output. After the last clock cycle, it has reached "out.10".

- (16) Define a clock function with the number of cycles as a parameter. The function is called "cycles".

```
(defun cycles (a)
  (repeat 1 1 a
    (h '(cl))
    (s '(x))
    (l '(cl))
    (s '(x))
  )
)

cycles
```

"defun" is one of the most useful functions in RNL LISP. It enables you to define your own functions that can subsequently be used in the same straightforward way as all other RNL functions. "cycles" illustrates this point. "defun" is followed by the name you give the function, which, in turn, is followed by a list of parameters representing the arguments to the function (see also (17)). The body of the function definition is made up of other functions or sequences of functions. In the case of "cycles", the body of the function definition is a "repeat" loop that is to be iterated a times. In the next paragraph you see how easily one can specify a sequence of a clock cycles with this newly defined function. (RNL has a clock function "c", which you could have used instead. We defined our own clock function here in order to illustrate the power of "defun". Another important function for you to explore with the help of the RNL User's Guide is "do".)

- (17) Set the input to the shift register to High and propagate it through the shift register for four cycles, using the "cycles" function. Then look at the outputs of latches 3, 4, and 5.

h in

done

(cycles 4)

Step begins @ 290 ns. ci=1 @ 0 in=1 @ 0 STATE AT END OF SIMULATION STEP: Current time= 290 in=1 out.10=0 ci=1

Step begins @ 290 ns. ci=0 @ 0 out.1=1 @ 1.2 STATE AT END OF SIMULATION STEP: Current time= 300 in=1 out.10=0 ci=0

Step begins @ 300 ns. ci=1 @ 0 STATE AT END OF SIMULATION STEP: Current time= 310 in=1 out.10=0 ci=1

Step begins @ 310 ns. ci=0 @ 0 out.3=1 @ 1.2 STATE AT END OF SIMULATION STEP: Current time= 320 in=1 out.10=0 ci=0

Step begins @ 320 ns. ci=1 @ 0 STATE AT END OF SIMULATION STEP: Current time= 330 in=1 out.10=0 ci=1

Step begins @ 330 ns. ci=0 @ 0 out.3=1 @ 1.2 STATE AT END OF SIMULATION STEP: Current time= 340 in=1 out.10=0 ci=0

Step begins @ 340 ns. ci=1 @ 0 STATE AT END OF SIMULATION STEP: Current time= 350 in=1 out.10=0 ci=1

Step begins @ 350 ns. ci=0 @ 0 out.4=1 @ 1.2 STATE AT END OF SIMULATION STEP: Current time= 360 in=1 out.10=0 ci=0

4

! out.3 out.4 out.5

out.3-H (vi=0.30 vh=0.90) (0.630400 pf) affects: input to functions for the following nodes: 39 39  
32 32

out.4-H (vi=0.30 vh=0.90) (0.630400 pf) affects: input to functions for the following nodes: 51 51  
44 44

out.5-L (vi=0.30 vh=0.90) (0.630400 pf) affects: input to functions for the following nodes: 63 63  
56 56

At the end of (16), all the cells in the shift register were at Low. After setting the input to High and applying four clock cycles, the High input should have arrived at "out.4". The function "cycles" made it very easy to apply the clock cycles. The result is as expected.

- (18) Open a log-file to record the activities of the RNL session. Define a node vector and change the report at the end of a simulation step using the vector.

```
(log-file "shift.log")
```

```
43363
```

```
(defvec '(bin path in out.1 out.2 out.3 out.4 out.5
           out.6 out.7 out.8 out.9 out.10))
```

```
(bin path <node in-H> <node out.1-H> <node out.2-H> <node out.3-H>
<node out.4-H> <node out.5-L> <node out.6-L> <node out.7-L>
<node out.8-L> <node out.9-L> <node out.10-L> )
```

```
(def-report '("State of Shift Path:" (vec path)))
```

```
("State of Shift Path:" (bin path <node in-H> <node out.1-H>
<node out.2-H> <node out.3-H> <node out.4-H> <node out.5-L>
<node out.6-L> <node out.7-L> <node out.8-L> <node out.9-L>
<node out.10-L> ))
```

```
5
```

```
Stop begins @ 360 ns.
```

```
State of Shift Path:
```

```
Current time= 370
```

```
path=0b11111000000
```

The "log-file" command has the effect that the file *shift.log* is opened and that all subsequent terminal activities will be recorded in this file. You can later analyze this file or edit it to keep parts of your interactive RNL session for inclusion in a control file (*.J* file). The number returned is the file identification (ID) of *shift.log*. You can close the log-file with the command (log-file nil). Alternatively, the log-file will automatically be closed when you exit RNL (we will close the log-file in this way).

The "defvec" command defines a data structure called a vector. A vector is a list of nodes. The value of a vector is determined by the value of its nodes. For example, if a vector has three nodes with the values 0, 1, 0 (Low, High, Low), the value of the vector would be 010 binary, or 2 decimal. There are a number of commands operating on vectors, e.g. assigning a value to a vector (see RNL User's Guide). The vector definition has the format (defvec '(base name node\_1 node\_2 node\_3 ...)). "base" is the base of a number system and can be bin for binary, oct for octal, hex for hexadecimal, and dec for decimal. When the value of the vector is printed, it is given as a number with the base given in "base" (see "def-report" below). "name" is the symbolic name of the vector, which is "path" in the example. "node\_1", "node\_2", "node\_3", ... are the nodes of the vector. In the example they are "in", "out.1", "out.2", ... The list returned after "defvec" is the internal representation of the vector.

The "def-report" specifies a new format for the report at the end of a simulation step, thus overriding the "def-report" given in the *shift.l* file (however, this does not influence the reports on value changes of nodes marked with "chflag"). With (vec path) you specify the inclusion of the vector "path" in the report (for more variations of the "def-report", see RNL User's Guide). A simulation step following the report definition illustrates the new report format. Since the base given in "defvec" is binary, the vector is printed as a binary number. The first two characters, 0b, indicate the base (they would be 0 for octal, 0x for hexadecimal, and none for decimal). The binary vector representation provides a good "visual" picture of the shift path. The input and the first four cells are High, the other cells are Low.

- (19) Open a file to store RNL output ("behavior" file) for subsequent printing on a Printronix printer. Then shift an input signal through the shift register and exit RNL. (The RNL output in the "behavior" file will be analyzed with the program MTP).

```
openplot "shift.beh"
```

```
nil
```

```
l in
```

```
(cycles 1)
```

```
h in
```

```
(cycles 1)
```

```
l in
```

```
(cycles 1)
```

```
(cycles 10)
```

```
exit
```

After the file *shift.beh* has been opened with the "openplot" command, information on all nodes whose change-flag is set will be stored in this file until the file is closed with "closeplot". Terminating RNL with "exit" will automatically close the file, in which case no "closeplot" is needed. ("openplot" returns "nil" after opening the plot file. The responses for the commands following "openplot" are reports similar to those already discussed and have therefore been omitted in the above text.)

Recall that we set the change-flag for "nodes" with the control file *shifts.l*. Later we interactively set the change-flag for "all\_nodes". Therefore, information on the following nodes will be stored in *shifts.beh*:

```
in  cl out.1 out.2 out.3 out.4 out.5
      out.6 out.7 out.8 out.9 out.10
```

To obtain a signal that can be easily identified in the printout, we apply a pulse at the input of the shift register by setting the input Low, High, and Low again, each followed by a clock cycle. Then we shift this pulse through the register with ten more clock cycles. With the last command we exit RNL.

#### 4.3. Printing RNL Output Using MTP

The program MTP has been developed for printing the output of RNL simulations on the Printronix dot matrix printer. MTP stands for Multiple Time-series Plot. From the behavior file produced by RNL at the end of the last section a printout of the signals can be obtained in three steps. You have to (a) create a file containing directives for MTP, (b) create a plot file and (c) send the plot file to the Printronix printer.

- (a) MTP has been designed to plot a number of different kinds of behavior files in a number of different formats. However, for the purpose of plotting the output of RNL only a few directives need be supplied. These are the following

##### START time

The START directive tells MTP when to start plotting (in nanoseconds). If not supplied its default value is 0. Data is skipped on the behavior file until an event is found whose time is greater than or equal to the START time.

##### STOP time

The STOP directive tells MTP when to stop plotting (in nanoseconds). STOP has no default value and must be supplied. If the STOP time is greater than the time of the last event on the behavior file, the plot will be concluded with the last event.

##### SCALE time

The SCALE directive tells MTP how many time units to plot per inch on the plot (nanoseconds / inch). The default value is 100.0 which is an appropriate value for RNL.

##### Signal selection and trace format directives

MTP does not plot every signal in the behavior file but only those that are specifically requested. This permits experiments which generate a large number of traces to be analyzed selectively. MTP provides several trace formats which can be used for analog and data domain values but the simplest and most useful for RNL is the LOGICAL format. To select signals A, B and C for plotting in LOGICAL format the necessary MTP directives are

Logical A  
Logical B  
Logical C

The order of the traces on the plot is determined by the order of the selection directives in the file. The first signal selected is plotted closest to the time axis. There can be a maximum of 20 signals selected on a given plot.

All MTP directives are case insensitive except for signal names and are free field, separated by blanks or CR.

For our example write the following into a directives file named *shift.dir*:

```
START 0.0
STOP 700.0
SCALE 100.0
LOGICAL in
LOGICAL cl
LOGICAL out.1
LOGICAL out.2
LOGICAL out.3
LOGICAL out.4
LOGICAL out.5
LOGICAL out.6
LOGICAL out.7
LOGICAL out.8
LOGICAL out.9
LOGICAL out.10
```

- (b) To create the plot file, which is to get the name *shift.plot*, enter:

```
% mtp shift.beh shift.dir shift.plot <CR>
```

MTP will provide information on its progress and echo the content of the directives file:

```
Select and preprocess input data
START 0.0
STOP 700.0
SCALE 100.0
LOGICAL cl
LOGICAL out.1
LOGICAL out.2
LOGICAL out.3
LOGICAL out.4
LOGICAL out.5
LOGICAL out.6
LOGICAL out.7
LOGICAL out.8
LOGICAL out.9
LOGICAL out.10
Sort preprocessed events
Generate the plot
```

Rasterize for the Printronix  
mtp complete, plot file is shift.plot

- (c) To send the plot file to the Printronix printer enter:

```
% lpr shift.plot <CR>
```

This will produce the printout shown on the following page. No signals are plotted before 370 ns (0.3700e+03), since we opened the behavior file only at that time. Remember, that at this time the outputs of the first four cells of the shift register were High, the other six outputs were Low. You can see this state of the register move through the output "out.10". You can also see the input signal move through the register immediately afterwards.

#### 4.4. Displaying RNL Output on a Graphics Terminal or Plotting it on an HP 7220 Plotter

The procedure for displaying RNL output on a graphics terminal or plotting it on an HP 7220 plotter is similar to that for printing it on the Printronix printer. Generate a plot file by substituting `gen_gps` for `mtp` in the commands given in paragraph (b) in the last section:

```
% gen_gps shift.beh shift.dlr shift.plot <CR>
```

However, `gen_gps` does not work properly with indexed nodes in its directives file.

Therefore, you have to remove all "LOGICAL" directives for indexed nodes in `shift.beh`, which leaves you with nodes "in" and "cl" only. If you want to trace the indexed nodes in the example of the shift register, you have to give them an additional non-indexed name with the "connect" command in the `net` file.

After you have created a `plot` file with `gen_gps`, enter

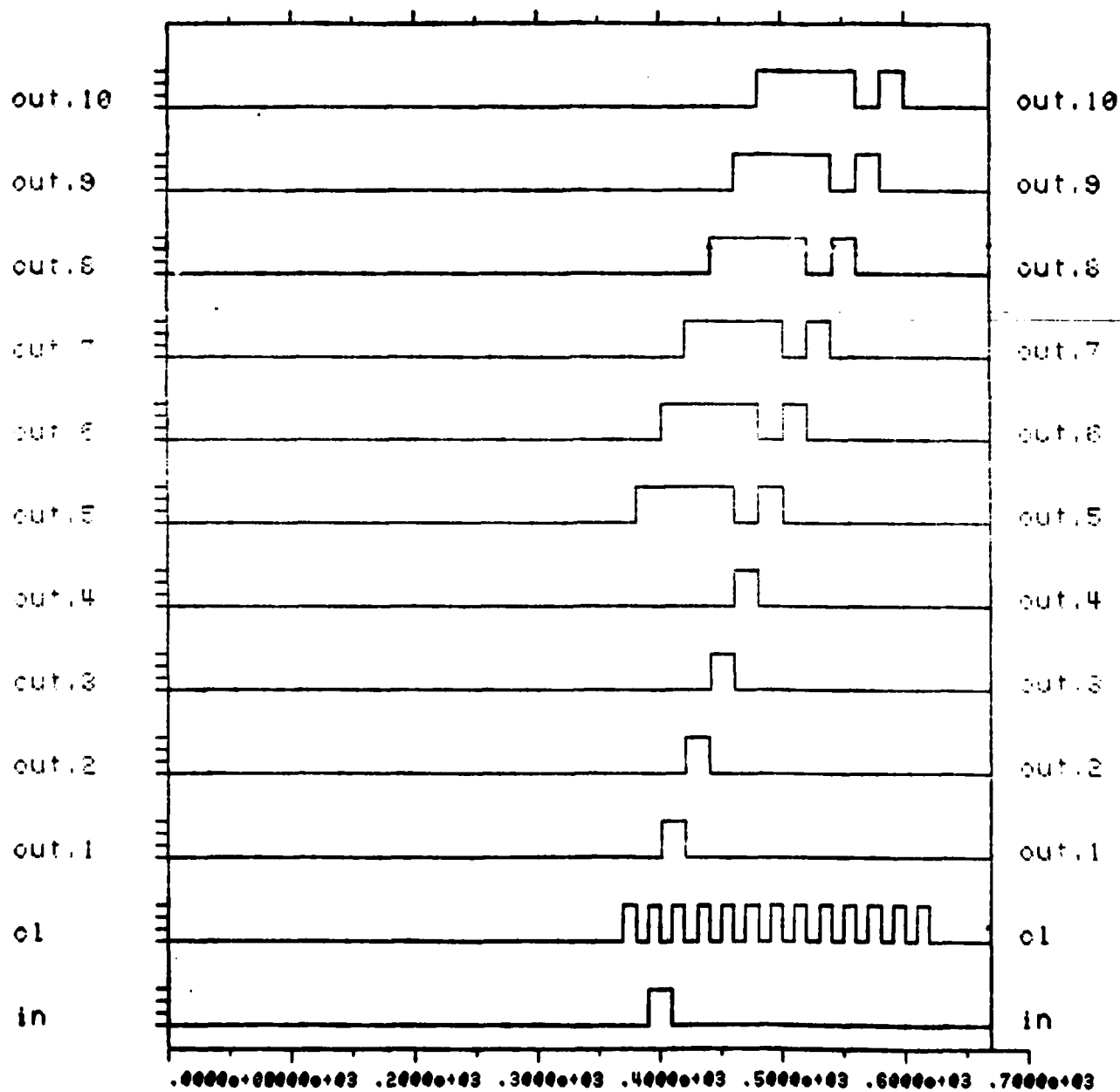
```
% gpplot shift.plot <CR>
```

to display on the GP19 terminal.

To plot on the HP 7220 plotter, you have to log in at the terminal next to the plotter (Sieg Hall, room 329), and enter

```
% hpplot shift.plot <CR>
```

(You will find some more details about these procedures in the UNIX on-line manual under "gps".)





## 5. Summary and Outlook

The simulation of the inverter and the shift register are examples of what you might encounter if you attempt to model and simulate your particular application with NETLIST and RNL. We were not able to look at all of the commands available in RNL and NETLIST, and therefore concentrated on some of the most frequently used ones. You will find complete lists of commands for both NETLIST and RNL in the references listed in the appendix.

The LISP-like command interpreter used by both NETLIST and RNL provides the facilities, and enables you to create your own special tools, for simulating very complex circuits. There are several ways to tackle the intricacies of RNL LISP. In addition to studying the User's Guides, you may work through examples of elaborate simulations, such as the simulation of the microcode sequencer referenced in the appendix. Another possibility is to have a close look at the function definitions given in the files *uwstd1* and *uwsim1*. Also, especially if you are fond of languages, you may want to study LISP in its "pure" form, without commands particular to RNL and NETLIST.

Whatever you do, keep in mind that RNL is a simulator based on a model of the real circuit, and therefore it is wise to know the assumptions underlying the model as well as the limits of its applicability. Information about the theory of NETLIST and RNL is provided in Chris Terman's original User's Guides and his thesis referenced in the appendix.

**Appendix 1 - Further References**

You may need information from the following sources if you want to use NETLIST or RNL more extensively.

From UW/NW VLSI Consortium, VLSI Design Tools Reference Manual:

1. NETLIST User's Guide (Contains, among other information, a list of all NETLIST commands.)
2. PRESIM User's Guide (Contains, among other information, specifications for the *config* file.)
3. RNL User's Guide (Contains, among other information, a list of all RNL commands.)

and in addition,

4. User's Guide to NET, PRESIM, and RNL/NL, Christopher J. Terman, M.I.T. Laboratory for Computer Science.
5. Simulation Tools for Digital LSI Design, (Thesis), Christopher J. Terman, M.I.T. Laboratory for Computer Science.
6. Simulating a Microcode Sequencer Using RNL: An Annotated Example of RNL Usage, Robert J. Fowler, UW/NW VLSI Consortium.
7. LISP, P.H. Winston, B.K.P. Horn, Addison-Wesley Publishing Company, 1981.
8. Metamagical Themas, The Pleasures of LISP: the chosen language of artificial intelligence, D.R. Hofstadter, article in three parts published in Scientific American, March and April 1983. RE LP

**Appendix 2 - Description of the .sim file of the example "Inverter" (section 3.1.2).**

The *inverter.sim* file produced in section 3.1.2. contains the following:

```
| units: 250.00 tech: ??? format: MIT
p in out Vdd 8.00 8.00 r 0 0 64.00
e in GND out 8.00 4.00 r 0 0 32.00
c out 3.000000e-02
```

Lines beginning with a vertical bar are considered comments by PRESIM, unless they have an entry "units:" or "format:". "units:" gives the conversion factor to centi-microns. "format:" is one of "MIT" or "UCB" (or, if no format is given, the old format originally used by the program is

assumed). "tech: ????" is the default comment indicating the technology used. You can change this comment with the `-t` option to NETLIST.

The following explanations relate to the MIT format, since the `.sim` file of the example has MIT format.

Lines 2 and three specify a p-channel and an n-channel transistor, respectively. Each of them is followed by the names of the nodes to which it is connected (gate, source, drain), by the length and width\* of the gate area in units of lambda, by a geometrical descriptor for the gate area which is always set to `r` (rectangular) by NETLIST, by its layout positional coordinates (NETLIST always specifies `0 0`), and finally by the gate area in square-lambdas.

The last line of `inverter.sim` specifies the output capacitance between `out` and `GND`, again with positional coordinates given as `0` by NETLIST, and a value of `0.03 pF`.

(For more details on the `.sim` file formats read the on-line manual with the UNIX command `man 5 simfile`).

### Appendix 3 - Preparation of a simple "config" File

The PRESIM User's Guide provides details on how to prepare configuration files. It lists all the possible parameters together with their default values. Generally, you need different configuration files for different technologies. Since we use a p-channel transistor, we need to use the configuration file to tell PRESIM the appropriate values for this device. The `config` used in the examples of this tutorial contains the following specifications:

```
resistance enh static 10 10 100000
resistance enh dynamic-high 10 10 10000
resistance enh dynamic-low 10 10 10000
resistance p-chan static 10 10 200000
resistance p-chan dynamic-high 10 10 20000
resistance p-chan dynamic-low 10 10 20000
```

### Appendix 4 - The "alias" file of the example "shift" (section 3.2.3).

The alias file `shift.al`, created by NETLIST in section 3.2.3. has only one line:

```
= in out.0
```

This tells PRESIM that "in" and "out.10" have been connected and therefore are considered as representing the the same node.

### Appendix 5 - Notice to Users of this Document

You are requested to direct your comments and questions relating to the form and content of

\* Note that the sequence of length and width is reversed in comparison to transistor specification with (ptrans ... ), (etrans ... ), etc.

this document to: Rudolf Nottrott, Systems Evaluation Group, UW/NW VLSI Consortium, Sieg Hall, FR-35, University of Washington, Seattle, Washington 98195. (Tel. (206) 545-2385, room 410).

# NETLIST/PRESIM/RNL - A TUTORIAL

*Robert Daasch  
Robert Fowler*

UW/NW VLSI Consortium  
Sieg Hall, FR-35,  
University of Washington,  
Seattle, WA 98195

## 1. INTRODUCTION

The following document is intended for beginning to intermediate users of the following programs;

- NETLIST,
- PRESIM,
- RNL.

Some familiarity with programming and the use of a computer terminal is assumed.

The approach used here is to provide examples that have been developed while using the programs here at the UW/NW VLSI Consortium. They are by no means exhaustive. Much of our attraction to these programs is their flexibility. This is particularly true in the RNL interpreter. When using this tutorial copies of the User's Guides (provided separately) should be available.

### 1.1. Ground Rules

The reader is encouraged to be sitting in front of a terminal that has these programs available. Much more can be learned by making the unavoidable mistakes when editing files and running these programs than by just reading. Error messages are at times cryptic and we make no effort here to wade through them. Readers are also encouraged to experiment and implement their own ideas. One very instructive method is to take these examples and modify and/or add extra capability. Learn by doing.

In sections where readers are expected to be editing; the text to be entered is in **bold face**. In the sections on the interactive use of RNL; user input will also in **bold face**. Program responses are in normal text. We recommend that an editor that supports Lisp (e.g. EMACS) be used if at all possible.

We make such a statement as both the NETLIST program and the RNL command interpreters are based on a Lisp syntax. That is to say program statements (commands) are surrounded with parentheses ( ). A general template for a command is

(command\_name arguments).

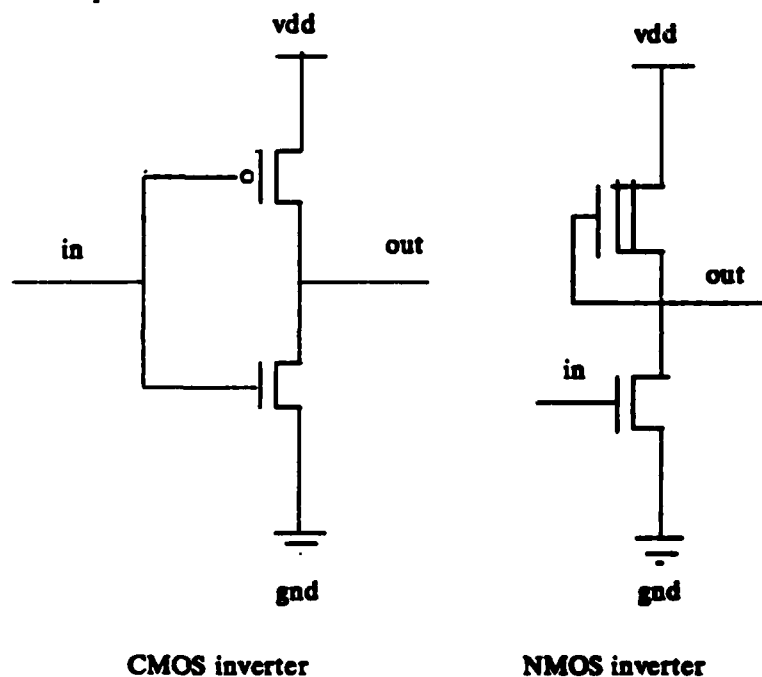
It should be assumed that all commands require the parentheses. It will be stated explicitly if they are not required.

## 1.2. Document Structure

We begin by discussing some of the basic statements used in writing NETLIST programs. This will be followed by the generation of the so-called .sim file of transistors. This file is the basic input for many simulators. Examples of using the program PRESIM, a .sim file preprocessor that generates input for the digital simulator RNL, are then presented. Simple interactive RNL experiments are shown and techniques for running RNL in batch mode are described. We end with some of the basic concepts of Lisp and a tour through some of the Lisp code that has been written locally to facilitate the use of RNL.

## 2. BUILDING A NETWORK DESCRIPTION USING NETLIST

For effective design it is important to establish that the design will work before layout is attempted. The program NETLIST allows the user to describe the circuit with a symbolic language. The NETLIST description is really a program which when run produces the list of transistors that make up the circuit. The following is a simple NETLIST program for a CMOS inverter. These commands will then be supplemented with others that will allow larger circuits to be partitioned.



### 2.1. Simple Commands

```

; All text following a semicolon is a comment and is ignored           ; (1)
; A CMOS inverter p type device 2X width of n device                   ; (2)
(node in out)                                                            ; (3)
(ptrans in out vdd 8 8)                                                  ; (4)
(etrans in gnd out 4 8)                                                  ; (5)
(capacitance out 0.83)                                                   ; (6)

```

- 1     As indicated by this line all text that appears after a semicolon (;) is considered a comment and is ignored by NETLIST.
- 3     This line declares the nodes *in* and *out*. You have to declare each node that you use. Nodes declared with the *node* command will be referred to as global nodes. Two global nodes that NETLIST knows about without your explicitly declaring them are *vdd* and *gnd*. Some programs are case sensitive and it is recommended you use them

as they are shown here. This redundant piece of information (after all, NETLIST can see that you are using a symbol as a node name when it builds the circuit) prevents spelling errors from causing unnecessary grief. Declarations are not as much trouble as they sound. Later a scheme will be presented that structures the NETLIST definition so that most nodes will be local to some module. Local nodes will be examined shortly. Using this technique only the few global nodes (usually clocks and i/o signals) have to be declared.

- 4 to 5 For simple circuits these are the two commands that do a lions share of the circuit description. They identify an individual transistor. They come in several types such as *etrams -> n type enhancements[0]*, *ptrans -> p type enhancements* and *dtrans -> NMOS depletion transistor*. There are others and the interested reader is encouraged to examine the NETLIST User's Guide to find out more. In most CMOS designs *etrams* and *ptrans* should suffice. The template for any of the transistor types is

(type gate source drain width length).

The type is as described above. The gate, source and drain arguments are the terminals of the transistor being declared. In line 4 the p type transistor is gated by node *in*, its source is the node *ow* and its drain is *vdd* (Note the case of *vdd* and *gnd* in lines 4 and 5). Source and drain in NETLIST is used solely to distinguish between the terminals of the transistor and do not imply anything about actual operating potentials. Width and length specify the size of the transistor. The values are given in a length parameter *lambda*. This allows for some technology independence in the network description. This unit is also used in layout programs. Typical values for *lambda* are 2-3 microns. In the inverter description above then, the n type transistor (line 5) is 1/2 the width of the p type.

- 6 Finally some capacitance is modeled on the node *ow* with the use of the capacitance command. The user is relieved of specifying the second terminal on the capacitor because all assumed to ground. The values (*0.03*) are in units of picofarads.

This file is then used as input to the program NETLIST. The actual running of this example is deferred momentarily as some additional NETLIST commands are investigated.

## 2.2. Additional Built-in Functions

Up to now we have used the transistor commands (*etrams* and *ptrans*) and the capacitance command. If this was all that was available life would indeed be tough. The general requirements for additional commands are function type, a technology and device sizes. Specification of the technology is important because NMOS uses depletion pullups whereas CMOS uses p type enhancements. This requires a slightly different handling of the signals. In NETLIST such commands exist and we will go through some now.

A CMOS inverter has the following template

(*cinvert out (ln width length)*)

*Cinvert* is the command name (like *etrams* above) and is followed by the argument declaring the output signal (*ow*). The next element of the command may look a bit strange but in conveys a lot of information. It is in fact a data structure we will be seeing often, the details of which are deferred to section 5 of this tutorial. For this example, it is declaring the input signal to be *in* and it defines the size (*width* and *length*) of the n type transistor in the CMOS inverter. This nearly satisfies our requirements but note that in the *cinvert* command (*ln width length*) only specified the size of the n type transistor. Where is the p device size declared?

---

[0] Historically NETLIST was written to describe NMOS circuits where there is just the one type of enhancement transistor.

This is a historical artifact of the NETLIST program. In NMOS the pullup is a depletion mode transistor which has its gate tied to the source. In the case of an NMOS inverter then the gate of the depletion device is in fact the output. (This is also true of nand and nor gates.) For the special case of NMOS design, we have a command that looks like

```
(invert (out width length) (in width length)).
```

As you can see the depletion pullup's size is declared on the output node. Similarly NMOS nand and nor gates have the same form[0]. In this context then the structure

```
(node_name width length)
```

specifies the size of the transistor gated the node `node_name`.

In CMOS the input gates both the p and n transistors. Moreover, nand and nor gates have equal numbers of pullup (p type) and pulldown (n type) transistors, the sizes of which could vary independently. Clearly some other solution must present itself.

The command `ratio` is the current solution to this dilemma. Its template is

```
(ratio value)
```

`Ratio` is the command name and `value` is a constant that is used to set the width of the p device. The p device's width is the product of `value` times the width of the n device ( $p_{width} = value * n_{width}$ ). The default for `value` is 2.0. The lengths of the two transistors are assumed equal. This doesn't not allow for complete independence of device size but has worked well in practice.

Returning to our need for a CMOS inverter command, we are left with the following

```
(node in out) ; (1)
(ratio 2.0) ; (2)
(cinvert out (in 4 8)) ; (3)
```

Of course we still need the `node` command as before. The last two commands are equivalent to the transistor commands discussed earlier.

```
(ptrans in out vdd 8 8)
(etrans in gnd out 4 8)
```

Its hard to see the gain with this example but if we consider the two possibilities for CMOS nand and nor gates the advantages start to present themselves. Within this scheme one could guess the commands for a 3 input nand to be,

```
(node out in1 in2 in3) ; (1)
(ratio 2.0) ; (2)
(cnand out in1 in2 in3) ; (3)
```

Again input and output nodes have to be declared with `node`. By dropping the width and length arguments for the inputs we have assumed default sizes for the enhancement transistors (2 lambda x 2 lambda). The `ratio` command sets the p devices to be two times the width of their corresponding n type just as before. The equivalent transistor description in this case is

[0] For example a complete specification of 2 input nand and nor gates.

```
(nand (out width length) (in1 width1 length1) (in2 width2 length2))
(nor (out width length) (in1 width1 length1) (in2 width2 length2))
```



getting quite large

```
(node out in1 in2 in3 1 2) ; (1)
(etrans in1 1 out) ; (2)
(etrans in2 2 1) ; (3)
(etrans in3 gnd 2) ; (4)
(ptrans in1 out vdd 4 2) ; (5)
(ptrans in2 out vdd 4 2) ; (6)
(ptrans in3 out vdd 4 2) ; (7)
```

1 to 3 If we explicitly describe the 3 input nand we have to declare 2 additional nodes. Nodes 1 and 2 are not particularly interesting as their only function is describing the connectivity to ground (*gnd*)

from the output node *out* (lines 2 and 3). Note when we used the built-in function they needn't be declared. NETLIST recognized the need for these "local" nodes and generated their names automatically. NETLIST always uses numeric node names for local nodes and the user is strongly advised to avoid their use in node declaration commands. In the next section we will see how these automatic nodes can be exploited even further.

5 to 7 The dual of the pulldown chain doesn't require any additional nodes but the 2 to 1 ratio in transistor width must be explicitly declared.

The same situation is encountered with CMOS nor gates[0]. Additional built-in functions of this nature are provided in NETLIST. The NETLIST User's Guide contains a brief description of each and in many cases contrasts the built-in function to its transistor equivalent.

### 2.3. User Defined Functions (Macros)

As shown in the previous section one of the main advantages of the built-in functions was the recognition and generation of the local nodes. The task of providing built-in functions for all the possible cases where they appear would be impossible but NETLIST provides an alternative. If user defined functions can be used as if they were built-in then specialized modules can be created as their need arises. An example of this would be if the device sizing of the gate functions was inappropriate for a design, a new function could be designed that fit the requirements.

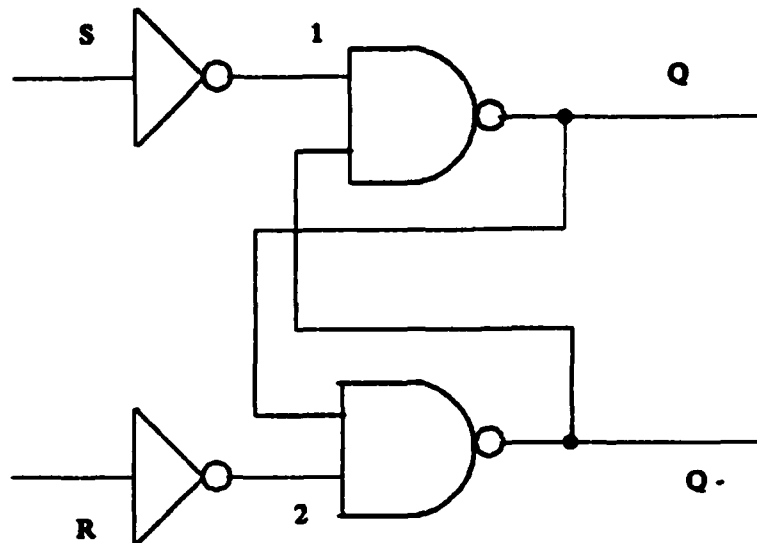
User defined functions are built in the form of macros. One way to think about a macro is a replacement for a related set of commands. Macros can have calling arguments (much like FORTRAN subroutines) and their own "local" nodes. Several examples of user declared macros will be presented in this section.

From a design point of view the macro of a SR latch shown below is not recommended. The choice was made to include an example where the circuitry needs little explanation so that the important features of the macro are evident. As has been the pattern important features are described on a line by line basis.

```
; CMOS SRlatch macro ; (1)
; Inverters are ratioed @ 2.0 ; (2)
; Nand gates are ratioed @ 1.0 ; (3)
(macro SRlatch (m_S m_R m_Q m_Q-) ; (4)
```

---

[0] A 3 input CMOS nor gate with default size n transistors is  
(cnor out in1 in2 in3)



SR latch

```

(local h1 h2)                                ; (5)
(ratio 2.0)                                  ; (6)
(cinvert h1 m_S)                             ; (7)
(cinvert h2 m_R)                             ; (8)
(ratio 1.0)                                  ; (9)
(cnand m_Q h1 m_Q-)                          ; (10)
(cnand m_Q- h2 m_Q)                          ; (11)
(capacitance h1 0.05)                        ; (12)
(capacitance h2 0.05)                        ; (13)
)                                              ; (14)

```

1 to 3    Comments just as before a begun with ";" and are ignored by NETLIST. They are useful especially as time goes by and you wonder what something really does.

4        This is the begining of the macro command. Its template is

(macro name (parameters))

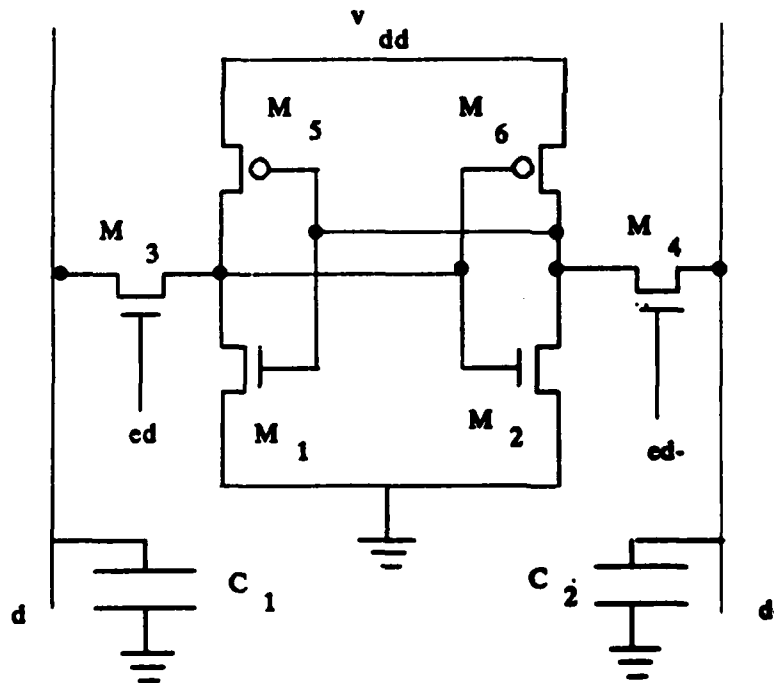
A macro is called with

(name (calling\_arguments))

Macros all begin with the left parenthesis and the word macro *"/macro"* followed by the macro's name (*SRlatch*). Following the name is the list of formal *parameters*. The number and type of *calling\_arguments* must match the formal parameters. As mentioned earlier this similar to the parameters in a FORTRAN subroutine. For example if a parameter is used as a node name in the macro's definition then when the macro is called the corresponding calling argument must be a declared node name. The only way encountered so far to declare a node name is with the use of node. Line 5 introduces another way to declare a legal node name.

- 5 Following the declaration of the macros name and parameter list, any local nodes that are needed are specified by the local command. Its form is exactly like the node command. This is useful because it helps in remembering what local does. It gives us a means of declaring the local names to a function and thus eliminates the need to specify them along with the important global signals (clocks, i/o, control) within the circuit. Every time the macro is called new nodes names are generated for these local nodes. Again we stress the fact that NETLIST uses numeric node names for these and one should avoid the use numeric node names in any other context.
- 6 to 13 This is the body of the macro. Any of the built-in functions can be used as a statement in the body. In addition any previously defined user macros can be statements in the body of the macro (shown in next example). Lines 7 and 8 also reflect that within the macro local nodes can be used just like the global nodes we have seen earlier. Lines 6 and 9 show the use of the rattle command. The scale values are set to 2.0 and 1.0 respectively. Upon exit from the macro the scaling value remains equal to 1.0. This can cause confusion and it is recommended that the rattle command be used frequently to ensure you are using the scale value you intended.
- 14 This right parenthesis ')' completes the macro. Careful inspection will show that this matches the left parenthesis used in line 4, the beginning of the macro declaration.

As a practical matter how and where does the use of the macro enter into the NETLIST program? For one thing the names in the parameter list must not conflict with the names of the global nodes. The following NETLIST program demonstrates the nesting of macros and hopefully provides a clear introduction to their use. We will also take this opportunity to introduce a command that allows repetition of a group of commands.



## 6 Transistor Memory

```

; macro for a 6 transistor memory cell, transistor sizing from      ; (1)
; "Analysis and Design of Digital Integrated Circuits,"            ; (2)
; David A. Hodges and Horace G. Jackson.                           ; (3)
(macro mem_cell (m_od m_od- m_d m_d-)                             ; (4)
  (local h1 h2)                                                    ; (5)

```

```

(rattle 0.125) ; (6)
(cinvert h1 (h2 16 2)) ; (7)
(cinvert h2 (h1 16 2)) ; (8)
(etrans m_ed m_d h1 4 2) ; (9)
(etrans m_ed- m_d- h2 4 2) ; (10)
) ; (11)
; end mem_cell ; (12)
; macro for generation of maxword x maxbit memory ; (13)
; requires macro mem_cell ; (14)
; bit and word index begins at 0 ; (15)
; bit and word index begins at 0 ; (16)
(macro mem_gen (maxword maxbit m_ed m_ed- m_d m_d-) ; (17)
  (repeat r 0 maxword ; (18)
    (repeat c 0 maxbit ; (19)
      (mem_cell m_ed.r m_ed-.r m_d.c m_d-.c) ; (20)
      (capacitance m_d.c 0.5) ; (21)
      (capacitance m_d-.c 0.5) ; (22)
    ) ; (23)
  ) ; (24)
) ; (25)
; end mem_gen ; (26)
; Begin main body of memory generation ; (27)
(node ed d ed- d-) ; (28)
(mem_gen 2 2 ed ed- d d-) ; (29)
; end main ; (30)
; (31)

```

1 to 11 The first macro declared is for a 6 transistor memory cell. The flip-flop is constructed from two cross-coupled inverters (lines 7 and 8). Note the use of local variables within the flip-flop. The enable lines control the reading and writing of bits or words into the flip-flop. In declaring the parameters to a macro one must use some care. Note in line 4 the prefix *m\_*. This is used as a mnemonic for formal macro parameters and limits the possibility of use the same name as global node name. Another point that should be made in this example is the use of a faction in the rattle command on line 6. This is perfectly reasonable when the drive must be very weak as is the case in such a memory cell. As noted in the comments a discussion of the transistor sizing can be found in "Analysis and Design..." The simulation of the 6 transistor memory cell is beyond the scope of this tutorial but is covered in section 4 of the RNL User's Guide. Finally, note the general formatting of the macro. The closing right parenthesis (line 11) is aligned with its corresponding left parenthesis. When writing macros some scheme like this should be employed. Some editors (EMACS) provide commands that will identify corresponding pairs of parentheses. As macro size increases use of an editor with this capability is strongly recommended.

14 to 25 This macro follows the same general scheme that has been outlined before. Important things to note here is the use of a new command and the nesting of one macro in another. When nesting one macro it is important to remember that a macro must be declared before it is executed. Line 18 provides a very useful capability when describing circuits. Repeat is much like the FORTRAN DO loop. It has the template

```

(repeat index initial final
  statements within repeat block
)

```

An index (line 18 *r* and line 19 *c*) is declared and followed by its initial and final values (e.g. 0 and *maxword* on line 18). As stated in the introduction, this command is contained within a set of parentheses (note: formatting can be very important during the debugging process). From the example it is obvious that repeat loops can be nested. Note how we used the repeat loop indices in constructing new node names (line 20). This is a very powerful feature in NETLIST programs. This ability gives us a third kind of node name, global and local detailed earlier and *structured*. A structured name has two or more components separated by periods. The first component must be a declared node name. As we have seen node names can be declared with a local or a node command. Another way is as it used here where the declared name comes from one of the parameters to the macro (Recall that node names used in calling a macro must be declared names.); the remaining components can be other names or expressions. Here we have used the indices from the repeat commands. You only have to declare non-numeric components of structured names -- for example, only "ed" must be declared, not "ed.1", "ed.2", etc. ( see line 29). Full details can be found in the NETLIST User's Guide.

to 30 Finally we see that the main body of NETLIST program has been reduced to 2 statements. Declaring the global node names and calling memory generator macro *mem\_gen*. Again note how a user defined macro is used just like the built-in functions described earlier.

### 2.3.1. Loading Macros

Up to now we have included all of the the macros that were required for the circuit descriptions in one file. However to use the *mem\_cell* and *mem\_gen* macros shown above it would be useful to be able to incorporate them into another design file without typing them in in their entirety. This can be done with the use of the load command. Its template is

```
(load "filename")
```

The actual filename is up to you but it must be surrounded by double quotes ". The practical effect is that any legal NETLIST commands and any the macros contained in the file are made available to the remainder of the program just as if they were typed in. An important requirement a macro must be loaded before it is executed. This is really the same requirement mentioned earlier, that a macro must be declared before it is used.

By putting the macros *mem\_cell* and *mem\_gen* in a separate file and with the load command, we could rewrite the program for generating the 2 x 2 memory.

```
(load "mem_primitives")           ; (1)
(node ed d ed- d-)                ; (2)
(mem_gen 2 2 ed ed- d d-)         ; (3)
```

Where the file *mem\_primitives* contains the NETLIST code for *mem\_cell* and *mem\_gen*.

### 2.4. Summary

This concludes the section on building circuit descriptions using NETLIST. We have discussed the following points

- Transistor descriptions (e.g. *etrans*, *ptrans*),
- Node name types (e.g. global, local and structured),
- Built-in functions (e.g. *cinvert*, *cnand*, *cnor*),
- User defined functions (i.e. macros),

- Repeating a blocks of NETLIST statements (i.e. repeat),
- Loading previously defined macros (i.e. load).

In following sections details of generating transistor files for simulation will be investigated. A flexible command language for simulation and an elaborate example of simulating a large design will be described.

There are many more functions that can be used in writing NETLIST programs. Moreover, arithmetic functions and common Lisp functions are available. These features are beyond the scope of this tutorial but users are encouraged to refer often to the reference manuals.

### 3. GENERATION OF PRESIM AND RNL INPUT FILES

The below is a session of running the `netlist` and `presim` commands in the C shell of UNIX[0]. Following the session, comments are indexed by line number.

```
% netlist inverter.net ; (1)
| units: 250 tech: ??? format: MIT ; (2)
p in out vdd 8.0 8.0 0 0 r 64.0 ; (3)
e in gnd out 8.0 4.00 0 0 r 32.0 ; (4)
c out 3.000000e-01 ; (5)
% netlist inverter.net -tisocmos ; (6)
| units: 250 tech: isocmos format: MIT ; (7)
p in out vdd 8.0 8.0 0 0 r 64.0 ; (8)
e in gnd out 8.0 4.00 0 0 r 32.0 ; (9)
c out 3.000000e-01 ; (10)
% netlist inverter.net -tisocmos -u200 ; (11)
| units: 200 tech: isocmos format: MIT ; (12)
p in out vdd 8.0 8.0 0 0 r 64.0 ; (13)
e in gnd out 8.0 4.00 0 0 r 32.0 ; (14)
c out 3.000000e-01 ; (15)
% netlist inverter.net inverter.sim -tisocmos ; (16)
% presim inverter.sim inverter ; (17)
Version 4.2 ; (18)
8 nodes; transistors: enh=1 intrinsic=0 p-chan=1 dep=0 low-power=0 pullup=0 resistor=0 ; (19)
Total transistors eliminated = 2 ; (20)
% presim inverter.sim inverter config ; (21)
8 nodes; transistors: enh=1 intrinsic=0 p-chan=1 dep=0 low-power=0 pullup=0 resistor=0 ; (22)
Total transistors eliminated = 2 ; (23)
% ; (24)
```

- 1 This command runs the program NETLIST. This is the simplest form of the command. That is, only an input file *inverter.net* is specified. The contents of *inverter.net* is the CMOS inverter described in the previous section. Output from *netlist* will be referred to as *.sim* file. With this usage of the command the *.sim* file (lines 2 to 5) is sent to standard output which in this case is the terminal.
- 2 A line that begins with a vertical bar (|) is generally a comment in the *.sim* file and is ignored. The only exception to this is when a *.sim* comment begins with a line of the form in 2. This line is required by programs in the Berkeley and UW tool sets. It contains a conversion number (units:) and a specification of the technology type (tech:) and finally the format: of the *.sim* file itself. Two formats are used within the

[0] UNIX is a trademark of Bell Laboratories and the C shell is a command shell generally used with the 4.1BSD UNIX.

UW/NW tools, the MIT format here and the UCB format produced by the layout extractor *mextra*.

- 3 to 5 Lines 3 and 5 represent the transistors in the inverter circuit. The MIT template for these "transistor records" is the following;

type gate source drain length width xpos ypos shape area.

Type identifies transistor records as n type enhancement (e)[0], p type enhancement (p) or when designing in NMOS depletion (d). Gate, source and drain are of course the terminals of the transistor. Xpos and ypos would provide the location of the transistor if the .sim file was extracted from a layout. In this case when only the connectivity is being declared these values are set to 0. Length and width represents the size of the transistor. When a .sim file is obtained from a layout and if a transistor is oddly shaped, they are approximate values. If an actual transistor is something other than rectangular, it is indicated with the shape field. A rectangular shape ("r") is assumed when the .sim file is generated by NETLIST. Area is the total gate area. Line 29 reflects load capacitance declared in the NETLIST input file (in picofarads).

- 6 In this command the additional input parameter -t has been declared. Note there is no space between the option flag and the technology name. As can be seen the output (lines 7 to 10) is substantially the same except with the technology now being declared as isocmos.
- 11 This time we have added another option which sets the centimicron value of lambda to 200 (i.e. lambda = 2.0 centimicrons). Again there is no space between the option flag (-u) and the value. This option is not reflected in the transistor sizes as both length and width are the same as the lines 7 through 10. Rather the units: parameter is now 200 rather than 250.
- 16 This is the usual form of the netlist command. Here we have specified an input file *inverter.net*, an output file *inverter.sim* and a technology *isocmos*. When an output file is desired it must appear as the second argument in the command.
- 17 The command *presim* prepares a binary file for input to RNL. Presim replaces the transistors in the .sim file with an equivalent sized resistors. The equivalent resistors are used with any capacitance on a node to compute an estimate of its transition delay time. The command must contain an input file (*inverter.sim*) and an output file (*inverter*). The defaults for this replacement are the source of no end of confusion for new users. The rule of thumb for the ratio of the conductivity of n type enhancements to p type enhancements is 2.0 to 2.5. Unfortunately, presim default assumes that all transistors have identical conductivity. To its credit presim does allow the user to parameterize the resistance and capacitance values by the use of the so-called config file (see PRESIM User's Guide and section 3 of the RNL User's Guide). The use of this file is shown on line 21 of this example.
- 19 to 20 Presim provides some additional output to the terminal. Line 19 contains information about the number of unique nodes encountered and a count, by type, of the transistors in the circuit. The node count of 8 is of no concern. The expected node count of 4 (*in*, *out*, *vdd* and *gnd*) has 4 nodes added to it that presim uses internally. Presim includes a network reduction scheme that improves execution speed during simulation. In effect it attempts to find a sum-of-products representation for as many of the nodes as possible. This has the practical effect of reducing the number of nodes (e.g. local nodes internal to the gate primitives) and transistors, line 20 reports the number of transistors involved in sum-of-products representations.

[0] The notation for the transistor types reflects the history of this simulator. It was originally designed with NMOS circuits in mind where there is but one type of enhancement transistor.

21 to 24 As mentioned earlier the configuration file is required when the resistor replacement operation needs parameterization. As can be seen the practical effect when issuing the `presim` command is to add an additional option (`config`). This option is the name of the configuration file you wish PRESIM to use. A extremely simple configuration file that sets the conductivity ratio of n to p type transistors to 2/1 is shown below;

```
resistance enh static 10 10 10000
resistance enh dynamic-low 10 10 10000
resistance enh dynamic-high 10 10 10000

resistance p-chan static 10 10 20000
resistance p-chan dynamic-low 10 10 20000
resistance p-chan dynamic-high 10 10 20000
```

The template is

```
resistance t-type r-type l w value
```

Resistance is a keyword for `presim`. T-type identifies the transistor type and r-type the resistance type. The three values shown are discussed in section 2 of the RNL User's Guide. L and w are the length and width of the transistor that has a resistance given by the last parameter (value).

### 3.1. Review

We have run two very important programs for the simulation of digital circuits using RNL. `Netlist` is a program that generates the transistor representation for the circuit. Frequently used options were outlined. Many more exist and interested readers are encouraged to read the NETLIST User's Guide.

`Presim` is a preprocessor for the digital circuit simulator RNL. It performs a transistor resistor replacement that is used by RNL to give estimates of the signal delay times. Its major pitfall is the default value for the resistance of p type transistors. A simple "workaround" was presented by the use of the configuration file. Again much more can be done with the configuration file and this information can be found in the PRESIM and RNL User's Guides.

## 4. INTERACTIVE SESSION WITH RNL

The following is an interactive session with RNL. It includes loading additional Lisp interface functions, formatting the results of a simulation step and running the inverter test case.

```
% rnl ; (0)
(load "awstd.l") ; (1)
done ; (2)
; (3)
(load "uwsim.l") ; (4)
Loading uwsim.l ; (5)
Done loading uwsim.l ; (6)
done ; (7)
; (8)
(read-network "inverter") ; (9)
; 8 nodes, transistors: enh=0 intrinsic=0 p-chan=0 dep=0 low-power=0 pullup=0 resistor=0 ; (10)
done ; (11)
; (12)
(def-report '("Current state:" newline in out)) ; (13)
```



```

("Current state:" "                                ; (14)
" in out)                                           ; (15)
                                                    ; (16)
(chflag '(in out))                                ; (17)
(in out)                                           ; (18)
                                                    ; (19)
l in                                               ; (20)
done                                              ; (21)
s                                                 ; (22)
                                                    ; (23)
Step begins @ 0 ns.                               ; (24)
in=0 @ 0                                          ; (25)
out=1 @ 0.6                                       ; (26)
Current state:                                   ; (27)
Current time= 100                                ; (28)
                                                    ; (29)
in=0 out=1                                       ; (30)
done                                              ; (31)
h in                                             ; (32)
done                                              ; (33)
s                                                 ; (34)
                                                    ; (35)
Step begins @ 100 ns.                            ; (36)
in=1 @ 0                                          ; (37)
out=0 @ 0.6                                       ; (38)
Current state:                                   ; (39)
Current time= 200                                ; (40)
                                                    ; (41)
in=1 out=0                                       ; (42)
done                                              ; (43)
(exit)                                           ; (44)
%                                                 ; (45)

```

- 0 From the C shell (note the "%" prompt) issue the command **rnl**.
- 1 to 7 From within the RNL command interpreter issue the commands (**load "uwstd.l"**) and (**load "uwsim.l"**). The two files **uwstd.l** and **uwsim.l** provide users of RNL with a simple command interface. In general these files will be loaded every time RNL is used. The interpreter when finished with a **load** command returns "done" (lines 2 and 7). Many of the commands return this.
- 9 The **read-network** command is issued next. This reads the file prepared by **presim** and in effect builds the network that is to be simulated. Recall when **presim** was run it provided a summary of the network. Similarly **read-network** returns a summary (line 10) however, the node and transistor count now reflects only those remaining after **presim**. For this example there were no internal nodes and the node count is unchanged. Again in line 11 "done" is returned by the command interpreter.
- 13 **Def-report** is a function provided in the **uwsim.l** package that allows users to format a report that is printed at the end of each simulation step. In setting a format up it is important that it begin with '(' and end with ')'. In this case we have a simple format that prints out the string "Current State:". A string of some kind is required the shortest one being the null string "". Following the string is a newline (newlines are interpreted as the sequence carriage return, linefeed). Then we request that the states of the nodes with the names *in* and *out* be printed. **Def-report** does not return

"done" like before but some rather odd text (line 15) that for the purposes of the tutorial can be ignored.

- 17 **Chflag** is another function provided in the `uwsim.l` package that informs the simulator which nodes should have their transition times reported. This is done by declaring a quoted list of symbols that have the same print names as the node you are interested in (e.g. *in* and *out*). Quoted lists in Lisp begin with `'` and end with `.`. Alert readers will note that the `def-report` in line 13 is also a quoted list. The list is returned (printed) by `chflag` when it is finished.

The arguments to the RNL command interpreter formally are known as Lisp symbols. In most cases, these symbols will have the same print names as nodes in the circuit (i.e. you type the same string). We have already seen examples of this in the setup phase detailed above. In the following we will refer to the arguments of many of the commands as nodes. This is only to avoid clutter, they are in fact Lisp symbols. A discussion of the evaluation of Lisp symbols etc. is deferred to section 5 of this tutorial. In areas of potential confusion we will refer to them as Lisp symbols. In this same vein all the commands described for the rest of this example are provided in the `uwsim.l` package. This should be assumed unless it is noted otherwise.

We are now in a position to do some simulation of the inverter. The commands that are presented here are simple. With some experience with the Lisp interpreter much more elaborate commands can be written. That is not the subject here however and is deferred.

- 20 **l** (*h* and *a* described later) is a command that sets all of the nodes listed (e.g. *in*) to a logic low. Note this and related commands do not require surrounding `()` when run interactively. This value will not change under any simulation conditions. Using this and related logic commands has the effect of declaring these nodes as inputs to the circuit.
- 20 to 25 **s** is the simplest simulation command available. It runs a single simulation step for a predetermined amount of time (default 100ns) or until the entire network being simulated has settled to a definite state. Recall in our set up phase we informed RNL (`chflag`) that the transition times of the nodes *in* and *out* should be reported. Lines 25 and 26 show these transition times relative to the current time reported in 24. Transitions of nodes that are set by **l** etc. are assumed to happen immediately hence the transition time of 0 in line 25 for node *in*.
- 27 to 31 When the simulation step is completed, **s** uses the `def-report` (line 13) to format the results of the step. Line 27 is of course the string we wanted to have printed, the current time (ns) is then given (this is always reported and need not be included in the format specification). Line 30 details the current states of all the nodes that were declared in the `def-report`. In this case *in* and *out* are indeed shown to be an inverted pair. "done" (line 31) is echoed to the terminal and this completes one simulation step. If a `def-report` was not specified a warning to that effect is displayed and then "done."
- 32 to 43 This sequence is very similar to the one just described. **h** is the analogous function for setting nodes to a logic high value. Not used in this example is the function **a**. It completes the set by declaring nodes to "unknown." The characteristics of nodes declared unknown are discussed in Section 4 of the RNL User's Guide. The results again show that the nodes of interest are indeed a inverted pair.
- 44 One exits the simulation by either typing on a newline the one of the strings (`exit`) or `exit` or by entering `CNTL D`. Exit is used here to return to the C shell of UNIX.

To release a node that has been declared an input with any of these commands one uses the **x** command followed by a list of nodes. Nodes that have been released will reflect their "true" state at the end of the next simulation step. The choice of **x** is a potential source of confusion as *X* (capital X) is used to represent the logic state of unknown. This is unfortunate but...

#### 4.1. RNL Control File

For small simulations such as the one described in the previous section all of input can be entered while in interactive mode. As simulations become larger or as one iterates on a design the need for a set of frequently used commands to be entered without retyping grows rather quickly.

RNL provides such a capability by the use of the control file. The commands in this file are executed upon entry into simulation. When the end-of-file is reached control is returned to the user. That is to say commands like the ones we have been through can be entered. What are typical commands put in the control file?

In the cases we have just analyzed the following file would reduce our efforts considerably.

```
; All text appearing after a semicolon is a comment and is ignored
; Rnl control file for the inverter example
(load "uwstd.l")
(load "uwsim.l")
(read-network "Inverter")
(chflag '(in out))
(def-report '("Current State:" newline in out))
```

This file is a collection of the setup commands that we issued first when running RNL interactively. For detailed analysis of these commands the reader is referred to the previous section. Note the use of the comment lines. Comments are begun with a semi-colon (;) and all text appearing after it until a newline is ignored by RNL.

##### 4.1.1. Using the RNL Control File

The only required modification to the in start up of RNL is to add the name of the file that contains the RNL commands and looks like,

```
% rnl control_file
```

where you can substitute any filename that suits you for control\_file. When using a control file RNL works rather quietly. Below you will see the output from the control file described above.

```
Loading uwsim.l
Done loading uwsim.l
; 8 nodes, transistors: enh=0 intrinsic=0 p-chan=0 dep=0 low-power=0 pullup=0 resistor=0
```

Clearly most of output that was generated interactively is gone. In fact we are left with only the fact that uwsim.l has been loaded and a summary of the preprocessed network from the read-network command. As shown below we can now set the input low and run a simulation just as was done before.

```
! in
done
```

```
s
```

```
Step begins @ 0 ns.
in=0 @ 0
out=1 @ 0.6
Current state:
Current time= 100
```

```
in=0 out=1
done
```

A useful technique to develop is one where the simulation experiment is verified interactively and is then followed by a larger simulation run in batch mode. This can be done with the use of the control file by adding simulation commands. Some care must be used here as some of the commands require a slightly different syntax when used in batch mode. For example when nodes are to be set at some input value (*h*, *l* etc.), in batch mode the nodes must be in the form of a quoted list (recall *'(, )*)

```
(h '(n1 n2 n3)),
```

the command to run a simulation step must also have the special quoted list, the empty list

```
(s '()).
```

This is the result of the RNL Lisp interpreter having a special form only for interactive commands. The discussion of this is in section 5.

#### 4.2. Useful Additions

To this point we have run through an example interactively and shown how one can condense the frequently used commands into an RNL control file. In this section, additional commands that can be used either interactively or through the control file will be explored.

##### 4.2.1. Buses

A very common situation in design is to have a group of signals that work together (i.e. buses). When doing simulation it would be most useful to be able to give these signals a name and refer to the whole set by that name. The *uwsim.l* package provides this through a set of vector commands. An example (albeit overly simple) of this would be to define the nodes *in* and *out* in the inverter example as a vector. This is done with the following command,

```
(defvec '(bin inoutvec in out))
```

The template for this command is

```
(defvec '(radix name list_of_nodes))
```

*Defvec* is the command name and notice that a familiar piece of syntax has appeared again, the quoted list (begins with *'* and ends with *)*). *Radix* is the number base that the vector will be printed in when you request that it be displayed. The choices are the familiar set, *bin* -> binary, *oct* -> octal, *hex* -> hexadecimal and *dec* -> decimal. *List\_of\_nodes* can be any number of nodes that defines the vector. In our case there are 2 *in* and *out*.

Vectors are a special data type that are composed of lists of nodes. There are functions provided for setting the vector's value (*lvec*) and finding out how a vector is defined (*vec-names*). These functions are detailed in section 8.2 of the RNL User's Guide. The handiest place for vectors however is when they are used in *def-report*.

```
; All text appearing after a semicolon is a comment and is ignored      ; (1)
; Rnl control file for the inverter example                             ; (2)
(log-file "in.log")                                                       ; (3)
(load "uwsim.l")                                                         ; (4)
(load "uwsim.l")                                                         ; (5)
(read-network "inverter")                                                ; (6)
(chflag '(in out))                                                       ; (7)
```

```
(defvec '(bin inoutvec in out)) ; (8)
(def-report '("Current State:" newline (vec inoutvec))) ; (9)
```

8 This control file is quite similar again to the one shown earlier but now on line 8 we have made use of the `defvec` command. In this case the radix has been set to binary and the name has been set to `inoutvec`. The signal names (nodes) `in` and `out` finish the definition of the vector.

9 The `def-report` command also reflects the use of `defvec`. Now the report uses `vec` which is defined to print the value of the named vector (e.g. `inoutvec`). Recall in declaring the vector we specified a binary radix. `Vec` reports the vector in this radix. This format replaces the individual node names and values we used earlier. Assuming that we have either given RNL a control file with this report or interactively declared a report the results of a step of simulation would be

```
l in ; (1)
done ; (2)
s ; (3)
; (4)
Step begins @ 0 ns. ; (5)
in=0 @ 0 ; (6)
out=1 @ 0.6 ; (7)
Current State: ; (8)
Current time= 100 ; (9)
; (10)
inoutvec=0b01 ; (11)
done ; (12)
```

11 The individual nodes and values have been replaced by the vector we declared. The name and the current value of the vector is reported. The prefix `0b` reflects the radix that was declared when the vector was defined. The other radix flags follow the UNIX convention (0 -> octal, 0x -> hex). One can also mix the printing of nodes and vectors thus

```
(def-report '("Current State:" newline in out (vec inoutvec)))
```

would be another way we could format the report.

#### 4.3. Event files

A very useful command for displaying and interpreting the results of your simulation is `openplot`. This command has the following template

```
openplot 'plot_file_name'
```

Note there are no parenthesis surrounding this command as there has been with the others that have been discussed. The effect of issuing this command is that all transitions that were requested with `chflag` are entered into the file `plot_file_name`. This file can then be used as input to programs that display the time series trace on either a 4010 compatible terminal, a HP 7220 plotter or on a Printronix raster printer. The details of the use of these programs are described in the man pages for `gps` and `mtp`.

At the end of the simulation the file is closed with

closeplot "plot\_file\_name"

Again the parentheses are not used.

#### 4.4. Local Network Walks

When working interactively, two commands that allow local areas of the network to be examined are presented. The first one allows the user to obtain all transistors that are gated by the node in question and any sum-of-products functions of which it is an input. This command will be referred to as a *forward reference*. The second command reports the list of transistors for which the node is either the source or drain and a summary of its sum-of-products representation if any. This will be referred to as *backward reference*. In the following example we will investigate the SR latch described earlier with these commands.

```

! S ; (1)
S=H [NOTE: node is an input] (vl=0.30 vh=0.80) (0.019200 pf) affects: ; (2)
input to functions for the following nodes: ; (3)
    1 ; (4)
    1 ; (5)
done ; (6)
? 1 ; (7)
1=L (vl=0.30 vh=0.80) (0.062800 pf) is computed from: ; (8)
CMOS ; (9)
    (nor (n-chan (and S=H )) resistance [1.00e+04, 1.00e+04] ; (10)
      (p-chan (and S=H )) resistance [5.00e+03, 5.00e+03] ; (11)
    ) ; (12)
done ; (13)
! 1 ; (14)
1=L (vl=0.30 vh=0.80) (0.062800 pf) affects: ; (15)
input to functions for the following nodes: ; (16)
    Q ; (17)
    Q ; (18)
done ; (19)
? Q ; (20)
Q=H (vl=0.30 vh=0.80) (0.012800 pf) is computed from: ; (21)
CMOS ; (22)
    (nor (n-chan (and 1=L Q=L )) resistance [2.00e+04, 2.00e+04] ; (23)
      (p-chan (and 1=L )) resistance [1.00e+04, 1.00e+04] ; (24)
      (p-chan (and Q=L )) resistance [1.00e+04, 1.00e+04] ; (25)
    ) ; (26)
done ; (27)
! Q ; (28)
Q=H (vl=0.30 vh=0.80) (0.012800 pf) affects: ; (29)
input to functions for the following nodes: ; (30)
    Q- ; (31)
    Q- ; (32)
done ; (33)
? Q- ; (34)
Q=L (vl=0.30 vh=0.80) (0.012800 pf) is computed from: ; (35)
CMOS ; (36)
    (nor (n-chan (and 2=H Q=H )) resistance [2.00e+04, 2.00e+04] ; (37)
      (p-chan (and 2=H )) resistance [1.00e+04, 1.00e+04] ; (38)
      (p-chan (and Q=H )) resistance [1.00e+04, 1.00e+04] ; (39)
    ) ; (40)

```

```

done                                     ; (41)
! Q-                                     ; (42)
Q=L (vl=0.30 vh=0.80) (0.012800 pf) affects: ; (43)
input to functions for the following nodes: ; (44)
    Q                                     ; (45)
    Q                                     ; (46)
done                                     ; (47)

```

- 1 to 6 This is the output of the forward reference command (/) for one of the inputs to the SR latch described earlier. Line 2 summarizes the parameters for node S. Several things should be observed here. First note that this node is considered an *input*. This is because the node has been set by of the command h. Similarly if l or u were used this messages would appear. The next parameters show the logic threshold values (voltage is normalized to 1.0) for the node. For a discussion of these values the reader is referred to section 2 of the RNL User's Guide. Next the total load capacitance is given in picofarads. Finally a summary of all nodes that this node is an input is reported. In this case only node l is affected. Recall that in building the SRLatch we used local nodes for the output of the inverters. The node l is one of the NETLIST generated nodes we talked about. If there were transistors that this node gated they would be reported next. The template for a transistor report is

type gate source drain resistance\_values

All of these parameters have their usual meanings (e.g. resistance in ohms).

- 7 to 12 We can continue the local network walk with a backward reference (?) for node l. Line 8 is similar to the forward reference but note how this node is not an input. It is computed from the function shown in lines 9 to 12. This is what a summary of a CMOS inverter looks like. The technology is indicated in line 8. This is followed by a list of each of the product chains in the sum-of-products description. Each product chain is prefixed with the type of transistor (n-chan or p-chan). In this case each product chain consists of just the one input S. At the end of each list of inputs a summary of the total resistance computed by PRESIM is given. The first of the two resistance values is used in computing the state of the node and the second is for estimating the delay time of the transition if any. Details are given in the RNL User's Guide in section 2. Again resistance values are given in ohms.
- 14 Pressing on with the forward reference l of node l we find that it is the input to the node Q (see line 17).
- 20 Node Q definition is then obtained with a backward reference ?. The function definition follows substantially the same form. The pulldown product term (it is the pulldown product because it is formed from n type transistors.) contains 2 inputs l and Q-. The pullup is composed of the first two term sum-of-products encountered that is, the two single term pullup products are ored together (l or Q-). This is the general form for nand gates. The dual of this combination of p and n transistors would be the pattern for a 2 input nor gate. Again each term is followed by its resistance values.
- 28 This is the forward reference for the node Q-. And here we find the effects of the cross-coupled nand gates forming the flip-flop in the SR latch. Note that Q- is a function of Q and vice versa (see line 45). Also the other local node from the second inverter is an input to Q-. Again NETLIST used numeric node names for these nodes and such names should be avoided elsewhere.

From this example then we have shown how the forward and backward reference commands can be used to get a feel for where a node lives in the circuit. This is particularly useful when portions of the circuit appear to be misbehaving. These commands can ensure that it indeed is "wired up" correctly.

#### 4.5. Summary

This completes an example for running RNL. This example is by no means exhaustive. In fact the Lisp command interpreter available in RNL makes the possibilities for elaborate simulation commands very attractive. Recall that one of the first things that we did during our session was to load the files `uwstd.l` and `uwsim.l`. These files represent some of the versatility of this interpreter. In the next section we will examine portions of these files for examples of how one might construct their own commands.

### 5. RNL LISP: SOME EXAMPLES

This section is a tutorial introduction by examples to programming in RNL Lisp. The particular examples chosen are taken directly from the package `uwsim.l`. They were chosen to be instructive not only from the standpoint of being examples of Lisp functions, but also because they are prototypical of the kinds of functions that a user may want or need to write. We are deliberately encouraging users to write their own user interface functions by copying these examples and modifying them.

#### 5.1. The RNL Lisp Interpreter.

In order to discuss the writing of Lisp functions we first present a brief introduction to the Lisp language and the Lisp interpreter. If at first you do not understand what is going on we recommend the following strategies: study the examples, reread this section carefully, and, most importantly, play with RNL interactively. Because Lisp is interactive you can often learn a lot more in a few minutes of hands on experimentation than in hours of staring at textbooks and manuals. The "real" Lisp systems that most resemble RNL Lisp are *MACLisp* (from MIT) and the *Franz Lisp* (from Berkeley). If you do consult other texts and manuals, ones that use either of these dialects will be the most useful.

It has been claimed that rather than standing for "LIST Processing" that Lisp is really an acronym for "Lots of Irritating Single Parentheses". Although the Lisp syntax is simple, elegant, and powerful, it has the unfortunate property that a user can easily wind up wasting time trying to balance parentheses and trying to understand poorly formatted Lisp code. There are two things that one can do to minimize this problem. The first is to use care in formatting your Lisp functions. Don't put too much on one line and use a consistent indentation style. The second thing that you can do is to use a screen-oriented text editor with a Lisp mode (e.g. EMACS) that helps you to keep track of the parentheses and indentation.

##### 5.1.1. Evaluation

Perhaps the most fundamental concept in Lisp is the *evaluation* of a Lisp object. Lisp objects are also known as *S-expressions* (The "S" stands for "symbolic".) and we will often just call them "expressions". The universe of Lisp objects is divided into two classes: primitive objects (also known as "atoms"), and lists.

In RNL Lisp there are several kinds of primitive object:

- symbols* are like variables in other programming languages. Each symbol potentially has a value, a functional definition, and a list of properties. In addition, it has a *print name* by which it is known, both on input and output. Usually print names are terminated by "white space" (spaces, tabs, and newlines), but print names containing these and other special characters can be entered by quoting the name with vertical bars (e.g. `long.name with white space!` is a symbol).
- numbers* can be integers or floating point numbers.
- strings* are pieces of text surrounded by double quotes (e.g. "this string").
- nodes* correspond to the electrical nodes of the circuit you are simulating. This is a data type not found in other dialects of Lisp. The print names of nodes can resemble symbols or numbers. If a symbol or number is used where a node is expected then RNL automatically tries to convert to the node with a similar print name. In addition,



nodes can be named as lists of the form *(-struct- a b c etc)* where each of *a, b, c, etc.* are either symbols or integers. Lists like this name nodes with names like *abc etc.* This allows you to create hierarchically structured naming schemes for your circuits. If you try to enter *abc etc* then the RNL interpreter will convert it to a list rather than a symbol, so if you want to have symbols with periods in their print names you have to use the vertical bar convention, e.g. *abc etc*.

A *List* is a sequence (a list) of Lisp objects that is bracketed with parentheses. Lists can and do contain other lists. For historical reasons the first element of a list is known as the "car" of the list and there is a function, *car* that extracts it from a list. The list formed by removing the car is known as the "cdr" (pronounced "koo-der") and is extracted using the function *cdr*.

Example: *((a b) (c (d)) e)*

This is a list of three elements. The first element (the car) is the list *(a b)*. The second element is the list *(c (d))*, and the last is the symbol *e*. The *cdr* is the list *((c (d)) e)*. Note that in the second element that *(d)* is a list of one element, not a symbol. Note also that the *cdr* of *((c (d)) e)* is the list *(e)* (i.e. *(c (d))* is one element of the original list). The *cdr* of *(e)* is the empty list *()*. The empty list is synonymous with the symbol *nil*.

That is all there is to Lisp data structures: atoms and lists. From these you build data structures, function definitions, and commands to the interpreter.

### 5.1.2. Evaluation

The central idea in the execution of Lisp programs is that of the *evaluation* of Lisp objects (expressions). Evaluation is a process that interprets a Lisp object and returns some other Lisp object, its value. The evaluation process can have side effects.

The evaluation of atoms is straightforward. When a symbol is evaluated the object the interpreter returns is the one that was most recently assigned to be the symbol's *value*. In other words, a symbol acts just like a variable in other programming languages. All the other types of atoms (numbers, strings, and nodes) are what is called "self-evaluating". That is, these objects and their values are identical.

The evaluation of lists is a little more complicated. In the interest of completeness we will give all the gory details here, but keep in mind that the simplest case is the most common. Usually lists are function calls, but sometimes they are what are called "special forms". In all cases, the *car* (first element) of the list being evaluated controls what will happen.

If the *car* of the list being evaluated is a *symbol* then the interpreter checks whether it has a function definition. If not, then the interpreter evaluates the symbol and uses the result as though it were the original *car* of the list.

If the *car* is an atom other than a symbol then this is an error because these types of atom cannot have function definitions.

If the *car* of the list being evaluated is itself a list then the interpreter first checks to see whether it is a function definition (see below). If it is not a function definition, then it is evaluated and the result is used as though it were originally the *car* of the list.

In this way, the interpreter repeatedly (recursively) evaluates the *car* of the list being evaluated until a function definition is found. Although this sounds complicated, the simplest case in which the *car* is a symbol with a function definition is the most common form and all other forms are extremely rare in circuit simulation applications.

### 5.1.3. Function Definitions

Function definitions come in two flavors. There are the built-in functions (including special forms) and there are user-defined functions. If you should print one of the former it would appear to be a funny symbol such as *#366506*. This indicates that the symbol is a built-in function or special form.

A user defined function definition appears as a list that looks like:

```
(lambda (x y) (+ 3 (* x y)))
```

The symbol *lambda* is a special symbol that means "This list is a function definition. Do not continue evaluation". The list "(x y)" names the symbols that act as the formal parameters to the function. The remainder of the list is the body of the function and is composed of a sequence of Lisp expressions (objects) that are evaluated in left-to-right order when the function is evaluated. The particular function defined above returns a value three greater than the product of its two parameters.

Thus far in our narrative the interpreter has reduced the car of the list it is evaluating to a function definition. If this is an ordinary function definition then the next step is the evaluation of the arguments to the function. Each of the remaining elements of the list is interpreted as an argument to the function. They are evaluated in left-to-right order and the objects returned are used as the values of the formal parameters of the function.

Finally, the body of the function is evaluated using these parameter values and the value returned is the last value returned when the body is evaluated.

Let us look in detail at the evaluation of the following expression:

```
((lambda (x y) (+ 3 (* x y))) 8 k)
```

The interpreter goes through the following steps:

1. First the car of the list is found to be a user-defined definition of a function with two parameters called *x* and *y*.
2. The next element in the list is evaluated. It is the integer 8 and is thus self-evaluating.
3. The last element in the list is the symbol *k*. Suppose that its value is 5.
4. Because *x* and *y* are the formal parameters of the function, their values are set to 8 and 5 respectively when the function is entered. When the function is exited the values they held previously will be restored.
5. The body of the function "(+ 3 (\* x y))" can now be evaluated:
  - a. The symbol + is found to have a built-in function definition and the first argument evaluates to 3.
  - b. The second argument to + is the list "(\* x y)". When it is evaluated a built-in definition is found for multiplication, *x* has the value 8, and *y* has the value 5. The expression "(\* x y)" therefore evaluates to 40.
  - c. Both of +'s arguments are now evaluated so the addition can proceed, returning 43. This completes the evaluation of the body of the function.
6. Since the last value returned in the evaluation of the body of the function is 43, this is also returned as its own value.

#### 5.1.4. Functions Versus Special Forms.

When a user-defined function is encountered as the car of a list all of the remaining elements of the list are evaluated and the results passed to the function as arguments. Furthermore, the number of formal parameters in the function definition and the number of arguments actually passed must agree.

The built-in function definitions do not have the same restrictions. Some functions can take a variable number of arguments. An example is the + function. It returns the sum of an arbitrary number of arguments. There are other built-in functions that do not evaluate one or more of their arguments. An example of this is the *setq* function that is used to set the value of a symbol. Evaluating the list

```
(setq sym1 (foo a b c))
```

has the side effect of setting the value of its unevaluated first argument (in this case, the symbol *sym1*) to the result of evaluating its second argument. This is also the value returned by the *setq* function.

Other symbols have built-in definitions that do not act at all like functions. These are called "special forms". For example, the *lambda* symbol is a special form that indicates function definition. Other special forms are used to define program control structures. While these special forms may return values analogously to functions, their interpretation is very different from that described above. For example, evaluating the list

```
(defun crunch (x y) (+ 3 (* x y)))
```

has the side effect of setting the function definition of the symbol *crunch* to the list:

```
(lambda (x y) (+ 3 (* x y)))
```

In this case none of the elements of the list are evaluated as arguments. (The result returned in this example is the symbol *crunch*.)

The *quote* special form is especially useful. It returns its argument without evaluating it. That is, if you want an object rather than its value you can use *quote* to inhibit evaluation. This is so useful that it has its own special alternate input syntax. The form *'obj* is translated to *(quote obj)* where *obj* can be any Lisp object. For example, if you type the expression:

```
(setq a "(x y z))
```

to the interpreter (note the two single quotes), then it will return the object:

```
(quote (x y z))
```

Because the self-evaluating objects (numbers, strings, and nodes) do not need to be quoted, they are also sometimes referred to as "self-quoting".

## 5.2. The "Top-Level Loop"

Now that we have introduced the data structures and the idea of evaluation in Lisp, we proceed to introduce the user interface to the RNL Lisp interpreter. The interface is called the "top-level loop" and consists of the following:

1. Read a Lisp object from the current input.
2. Evaluate the object read in step 1.
3. If the current input is the terminal then print the result of step 2.
4. Go to step 1.

The input to the interpreter is buffered on a line by line basis. Thus, RNL does not see anything you have typed until you enter a "newline" and you can use the standard "within line" editing of the system keys to modify the input before you enter it. Even though you have completed a line, RNL might not have read a complete Lisp object and therefore might not respond to you. For example, suppose you enter the lines

```
(+ 3
 (* 8 5))
43
```

(We use the convention that user input is displayed in bold type.) After you enter the second line RNL responds by evaluating the expression and printing the result, the number 43. Note that it did nothing after you entered the first line because it had not read a complete Lisp object at that point.

In order to reduce the number of parentheses you have to type, RNL Lisp has a special alternative syntax you can use. If the first thing on a line is a symbol then RNL interprets it as a function name, creates a quoted list out of the rest of the line, and passes that list as the only argument to the function. For example, the following two lines of input are

interpreted identically because the "reader" of the top-level loop converts the second line into the first line. (Length counts the number of elements in a list.)

```
(length '(a b c (4 5) 3 25 8.0E6))
7
length a b c (4 5) 3 25 8.0E6
7
```

You should remain aware that if you are using this alternative syntax that the entire command must be on a single line. While this is convenient for invoking some kinds of functions it does make it awkward to find the value of a symbol. Consider the following sequence:

```
(setq a 23)
23
a
;illegal function object
23
(eval 'a)
23
```

Because of the alternate syntax, when we tried to get the interpreter to evaluate the symbol *a* as an expression, the "reader" actually created the list

```
(a '())
```

and that is what was evaluated. The function *eval* evaluates its argument an extra time, so passing a quoted symbol to it results in only one evaluation being done.

The discussion of evaluation in the previous section covers the case in which everything works. Inevitably, however, there will be errors such as the one above in which no function definition was found for the symbol *a*. Whenever an error like this is detected an error message is printed and all current attempts to evaluate Lisp objects are aborted. This leaves you back in the top-level interactive "read-eval-print" loop. A particularly troublesome aspect of this is that if an error is detected while you are using the *load* function to read a command file that the *load* itself is terminated by the error so that the remainder of the file will not be read.

Note that the result of evaluating a command is printed only when input is being taken from the standard input. This may make it difficult to locate errors in command files that are being "loaded".

## 6. EXAMPLES FROM *uwsim.l*

To make things clearer we proceed to look at some examples taken directly from *uwsim.l*. These examples are augmented with line numbers in square brackets at the left. These numbers are not part of the code but have been added for the purposes of this presentation.

While a large part of most command files is the definition of functions and data structures that will be used later, part of all command files is the initialization of "global" symbols that parameterize those functions.

```
[1] (setq incr 1000)
[2] (setq switch-level nil)
[3] (setq relative-timing t)
```

These three commands use the *setq* function to set the default values of parameters that are used to control your simulation. Line 1 sets the value of the symbol *incr* to 1000 RNL time units (100.0 nano-seconds). Line 2 ensures that the simulator will use the RC model for simulation rather than the unit delay switch model and line 3 sets a flag that forces the *stop* function (see below) to report simulation times relative to the start of the simulation step. These lines illustrate the concept of self-evaluating objects. The number 1000 evaluates to itself.

The two symbols `nil` and `t` are predefined by the Lisp interpreter so that they are their own values. In addition, `nil` has the semantics of being identical to the empty list, `()`.

;; run a simulation step and print a report at the end

```
[1] (defun s (dummy)
[2]   (step incr)
[3]   (wr-report)
[4] )
```

This example illustrates the definition of the function `s` used in section 4. The function `s` does not add any real power to the user interface, rather it serves the purpose of reducing the amount of typing you need to do. This runs the simulation for a time increment of the current value of the symbol `incr` and then calls the function `wr-report` with no arguments to write out a summary report at the end of the step.

Note that `s` has a dummy parameter called `dummy`. This is because `s` is intended to be used interactively using RNL Lisp's alternate syntax. When you type `"s"` on a line by itself without any parentheses, the reader converts it to the list `(s '())`. In order for `s` to work correctly it must expect to see one argument even though that argument will be ignored.

```
(defun _prinum (base num) ; only called to bind the right val to base
  (princ num))
```

The function `_prinum` is used by various other functions in the `nwsim.l` package. Its job is to print an integer in the radix specified by `base`. The Lisp printing routines use the current value of `base` to control the radix used for output. By naming one of the parameters of `_prinum` `base` we use the argument binding mechanism of RNL Lisp to temporarily change the value of `base` to the desired value. When `princ` is called, this is the value it uses. When `_prinum` returns the previous value of `base` is restored.

This example illustrates the principle of dynamic variable scoping in Lisp. Programming languages such as C or Pascal have what is known as a textual, or static, scope rule for the use of local variables. That is, the part of a program that sees a particular local variable is statically limited to the text of the routine in which it is defined. In contrast, Lisp uses a dynamic scope rule. When a Lisp special form (e.g. a function definition) uses a symbol as a "local variable" (e.g. a parameter), then the old value of the symbol is saved away and is restored only when the special form returns. Any functions that are called before the special form returns will see the new value of the symbol. Furthermore, any changes made to the value of the symbol will be lost when the old value is restored.

; chflag sets the STOPONCHANGE flag for the nodes in l

```
[1] (defun chflag (l)
[2]   (do ((here 1 (cdr here)))
[3]       ((null here) l)
[4]       (stop-on-change (car here) t)
[5]   ))
```

The function `chflag` takes as an argument a list of nodes. It sets the STOPONCHANGE flag for each of the nodes in the list. We use it as an example to introduce the `do` special form and, in particular, to show how a `do` is typically used to perform a function on each of the elements in a list. Note that although `chflag` has just the one argument (the list `l`), the list itself is not of fixed length.

Line 1 defines `chflag` to be a function with a single argument called `l`. Line 2 begins a `do` form. A `do` is a generalized iteration construct that allows you to define multiple local symbols to be used in the iteration. While a `repeat` increments its single local symbol on each iteration, a `do` allows arbitrary computations to be done on to get the new values of its local symbols.

The first thing that follows the `do` is the list of local symbol declarations. In this case there is only one element in that list, the declaration of *here*. A declaration is a list of one to three elements. The first element is a symbol, in this case *here*. The second (optional) element of the declaration is evaluated to get the initial value for the symbol, in this case returning the value of 1. The third (optional) element in the declaration list is an expression that is evaluated at the beginning of each successive iteration and whose result then becomes the value of the symbol at the start of that iteration. In this case it is "(cdr here)". Thus, *here* is defined to initially be the original list of nodes and on each successive iteration *here* is shortened by dropping the current first element.

The next thing in a `do` following the declarations list is an "exit clause". This is on line 3. An exit clause consists of a list of expressions. The first expression (sometimes called the predicate) in this list is evaluated at the start of each iteration. In this case it is the expression *(null here)*. In Lisp "true" means "anything other than *nil*". When a built-in function has to return a value meaning "true" it uses the symbol *t*, but any non-*nil* Lisp object will do. When the value of the predicate of the exit clause becomes true (think of it as "non-*nil*") then all of the expressions in the rest of the exit clause are evaluated in left-to-right order and the loop is exited, returning the value of the last expression in the exit clause. In the example, the predicate *(null here)* will be true when *here* becomes *nil* (the null or empty list), that is, when we've removed all of the node elements. The last expression in the exit clause is the symbol *t*, so the loop returns its value. Since the `do` is the last (only) expression in the definition of the `chflag` function, the value of *t* is also the value returned by it.

The remainder of the `do` form is a sequence of expressions that are known as the body of the loop. The expressions of the body are evaluated on each iteration in which the predicate of the exit clause is *nil*. In the example the body is the single expression that is a call to the primitive function `stop-on-change` with the *car* (its first element) of *here* as the first argument and with *t* as the second. This has the effect of flagging the node so that the simulation is halted whenever the node changes state so that the event can be reported or some other special action can be taken.

; Run a simulation step, reporting transitions

```
[1] (defun step (incr)
[2]   (printf "Step begins @ %S ns.0 (/ (float current-time) 10.0))
[3]   (do ((stop-time (+ incr current-time)) (savex (* current-time 1))
[4]       (n t))
[5]       ((null n))
[6]       (setq n (cond (switch-level (switch-step stop-time))
[7]                     (t (sim-step stop-time))))
[8]       (cond (n (dpy-node-trans n)
[9]               (printf "@ %S0
[10]                  (/ (cond (relative-timing (- current-time savex))
[11]                        (t (float current-time)))
[12]                  10)))
[13]       (t nil)))
[14] )
```

The function `step` is interesting to look at for a couple of reasons. It is the function that one uses to simulate a circuit for a particular time increment and is therefore worth knowing about both for having an understanding of what is going on and also in case one wants to modify it. It also contains a more complicated example of a `do` as well as introducing the `cond` special form. The first thing done (line 2) is to print out the current elapsed simulated time in nanoseconds.

The RNL provided simulation primitives (`sim-step` and `switch-step`) both operate by advancing the simulated time until either the specified stop time is reached (returning *nil* in this case), or until some circuit node that has been flagged changes state (returning the node

that changed). The function `step` uses these primitives to run the simulation for the fixed time increment specified in the parameter `incr`. Once the beginning time is printed, the body of the function is loop implemented with a `do`. On lines 3 and 4 the local variables are defined. `Step-time` is initialized to be the time to stop the simulation step, `savex` is initialized to be the time at which the step started, and `n`, which will be used to hold the value returned by each call to the appropriate simulation primitive, is initialized to `t`. Since the simulation primitives return `nil` when the appropriate stop time is reached, (null `n`) is used as the predicate (line 5) of the exit clause for the loop.

Lines 6 and 7 are the call to the simulation step. A `cond` special form consists of the symbol `cond` followed by a number of lists of expressions known as clauses. These are similar to the exit clause of a `do`. In a `cond` the clauses are examined in left-to-right order. For each clause, the predicate is evaluated and if it is true (returns other than `nil`) then the rest of the expressions in that clause are evaluated and the `cond` is exited, returning the value of the last expression evaluated. In the `cond` on lines 6 and 7, if `switch-level` is not `nil` then `switch-step` is called, otherwise (since the value of `t` is `t`) `sim-step` is called. Since the value returned by a `cond` is the last value computed in the clause that is executed, the value returned by this `cond` will be the value returned by the appropriate simulation primitive. This value is either a node with its `STOPONCHANGE` flag set or `nil` and the `setq` on line 6 makes it the value of `n`.

The first clause of the `cond` on line 8 is executed if the value of `n` is not `nil`. It first passes `n` to the function `dpy-node-trans` and then prints the current time, either relative to the start of the step or in absolute terms, depending on the value of the symbol `relative-timing`. The default clause (line 13) is included just for readability.

To summarize, the body of `step` repeatedly calls a simulation primitive until that primitive returns a `nil` to indicate that the desired termination time has been reached. Each time the simulation primitive does not return `nil` it is because a flagged node changed state, and a reporting function is called.

```
;; run n clock cycles
[1] (defun c (n)
[2]   (cond ((eq n nil) (setq n 1))
[3]         ((not (numberp n)) (setq n (car n))))
[4]   (do ((index 0 (1+ index))
[5]       (i 0))
[6]       ((= index n) (wr-report))
[7]       (set-node 'phi1 1)
[8]       (set-node 'phi2 0)
[9]       (step incr)
[10]      (set-node 'phi1 0)
[11]      (step incr)
[12]      (set-node 'phi2 1)
[13]      (step incr)
[14]      (set-node 'phi2 0)
[15]      (step incr))
[16]   )
```

The function `c` runs the simulation for one or more cycles of a two phase (four interval) non-overlapping clock defined using the predefined node names "phi1" and "phi2". This is written to take advantage of RNL Lisp's alternate input syntax. The symbol `n` will be the number of cycles to simulate.

The `cond` in line 2 is used to set `n` to the correct value. The first clause sets the count to 1 if the arguments to `c` is the empty list (`nil`) as would be the case if the command were entered by typing "c" on a line by itself. The second clause is used to differentiate between the cases of calling `c` using the standard parenthesized syntax (e.g. "(c 5)") or the interactive syntax (e.g. "c 5" on a line by itself).

The rest of the body of `c` is a simple `do` loop that uses `index` as the loop index to count a clock cycles and call `wr-report` when it exits. Lines 7 through 15 handle the mechanics of raising and lowering the clock lines as appropriate and advancing the simulated time by calling `step`. If the circuit you are simulating uses a different clocking discipline then you should write your own "clock cycle" function analogous to `c`. The code for `c` provides a reasonable template to modify for your own needs.

```

; unchanged-since returns a list of the nodes that are unchanged since time.
[1] (defun unchanged-since (time)
[2]   (prog (uc-list)
[3]     (walk-net '(lambda (n)
[4]                 (cond ((< (node-time n) time)
[5]                       (setq uc-list (cons n uc-list)))
[6]                       (t uc-list)
[7]                 )))
[8]   )))

```

The function `unchanged-since` introduces some new Lisp constructs and also illustrates the usefulness of the `walk-net` primitive when you want to examine all the nodes in your circuit. `Unchanged-since` returns a list of the nodes whose last transition occurred prior to the argument `time`.

The `prog` special form is used to define local symbols. After the symbol `prog` comes a list of symbols that are to be made local and following that list is a sequence of expressions. When a `prog` is evaluated the old values of the symbols in the list are saved, their current values are set to `nil`, and the sequence of expressions is evaluated. When the evaluation is done the old values of the local variables are restored and the result of the evaluation of the last expression in the sequence is returned as the value of the `prog`. In this case, the only local symbol is `uc-list` and the value of the `prog` will be the result of evaluating the `walk-net` function.

The `walk-net` function takes as its argument a function definition or a symbol that has a function definition. `Walk-net` then traverses the circuit and for each node it passes that node to the function and evaluates it. The value returned by `walk-net` is the value returned by the function the last time it was called.

In the example we didn't want to bother with giving a symbol a function definition, so we used the `lambda` special form to define an "anonymous" function definition. This anonymous function will build a list of nodes that have not been recently changed and keep it as the value of `uc-list`.

The predicate of the first clause of the `cond` beginning on line 4 tests (using the `<` predicate) whether the most recent node transition time of the parameter `n` was before the threshold `time`. If so, then it sets the value of `uc-list` to be the list consisting of `n` as the first element followed by the the list that was the previous value of `uc-list`. This is done by the `cons` function. Note that this returns the list as its value. If the node has been recently changed then the "default" clause (line 6) of the `cond` is executed, returning the unchanged value of `uc-list`. The function defined by the `lambda` thus always returns the current list. `Walk-net` therefore will return the final value of the list and therefore so will the `prog` and therefore so will `unchanged-since`.

## 6.1. Summary

We hope that by presenting these examples in detail that we have made a little less imposing the prospect of writing your own Lisp functions as part of your RNL simulation.



# SPICE User's Guide

*A. Vladimirescu, Kaihe Zhang,  
A.R. Newton, D.O. Pederson, A. Sangiovanni-Vincentelli*

Department of Electrical Engineering and Computer Sciences  
University of California  
Berkeley, Ca., 94720

This manual corresponds to SPICE version 2G6

**Acknowledgement:** Dr. Richard Dowell and Dr. Sally Liu have contributed to develop the present SPICE version. SPICE was originally developed by Dr. Lawrence Nagel and has been modified extensively by Dr. Ellis Cohen.

SPICE is a general-purpose circuit simulation program for nonlinear dc, nonlinear transient, and linear ac analyses. Circuits may contain resistors, capacitors, inductors, mutual inductors, independent voltage and current sources, four types of dependent sources, transmission lines, and the four most common semiconductor devices: diodes, BJT's, JFET's, and MOSFET's.

SPICE has built-in models for the semiconductor devices, and the user need specify only the pertinent model parameter values. The model for the BJT is based on the integral charge model of Gummel and Poon; however, if the Gummel-Poon parameters are not specified, the model reduces to the simpler Ebers-Moll model. In either case, charge storage effects, ohmic resistances, and a current-dependent output conductance may be included. The diode model can be used for either junction diodes or Schottky barrier diodes. The JFET model is based on the FET model of Shichman and Hodges. Three MOSFET models are implemented; MOS1 is described by a square-law I-V characteristic MOS2 is an analytical model while MOS3 is a semi-empirical model. Both MOS2 and MOS3 include second-order effects such as channel length modulation, subthreshold conduction, scattering limited velocity saturation, small size effects and charge-controlled capacitances.

## 1. TYPES OF ANALYSIS

### 1.1. DC Analysis

The dc analysis portion of SPICE determines the dc operating point of the circuit with inductors shorted and capacitors opened. A dc analysis is automatically performed prior to a transient analysis to determine the transient initial conditions, and prior to an ac small-signal analysis to determine the linearized, small-signal models for nonlinear devices. If requested, the dc small-signal value of a transfer function (ratio of output variable to input

source), input resistance, and output resistance will also be computed as a part of the dc solution. The dc analysis can also be used to generate dc transfer curves: a specified independent voltage or current source is stepped over a user-specified range and the dc output variables are stored for each sequential source value. If requested, SPICE also will determine the dc small-signal sensitivities of specified output variables with respect to circuit parameters. The dc analysis options are specified on the `DC`, `TF`, `.OP`, and `SENS` control cards.

If one desires to see the small signal models for nonlinear devices in conjunction with a transient analysis operating point, then the `.OP` card must be provided. The dc bias conditions will be identical for each case, but the more comprehensive operating point information is not available to be printed when transient initial conditions are computed.

### 1.2. AC Small-Signal Analysis

The ac small-signal portion of SPICE computes the ac output variables as a function of frequency. The program first computes the dc operating point of the circuit and determines linearized, small-signal models for all of the nonlinear devices in the circuit. The resultant linear circuit is then analyzed over a user-specified range of frequencies. The desired output of an ac small-signal analysis is usually a transfer function (voltage gain, transimpedance, etc). If the circuit has only one ac input, it is convenient to set that input to unity and zero phase, so that output variables have the same value as the transfer function of the output variable with respect to the input.

The generation of white noise by resistors and semiconductor devices can also be simulated with the ac small-signal portion of SPICE. Equivalent noise source values are determined automatically from the small-signal operating point of the circuit, and the contribution of each noise source is added at a given summing point. The total output noise level and the equivalent input noise level are determined at each frequency point. The output and input noise levels are normalized with respect to the square root of the noise bandwidth and have the units Volts/ $\sqrt{\text{Hz}}$  or Amps/ $\sqrt{\text{Hz}}$ . The output noise and equivalent input noise can be printed or plotted in the same fashion as other output variables. No additional input data are necessary for this analysis.

Flicker noise sources can be simulated in the noise analysis by including values for the parameters `KF` and `AF` on the appropriate device model cards.

The distortion characteristics of a circuit in the small signal mode can be simulated as a part of the ac small-signal analysis. The analysis is performed assuming that one or two signal frequencies are imposed at the input.

The frequency range and the noise and distortion analysis parameters are specified on the `.AC`, `NOISE`, and `.DISTO` control lines.

### 1.3. Transient Analysis

The transient analysis portion of SPICE computes the transient output variables as a function of time over a user specified time interval. The initial conditions are automatically determined by a dc analysis. All sources which are not time dependent (for example, power supplies) are set to their dc value. For large-signal sinusoidal simulations, a Fourier analysis of the output waveform can be specified to obtain the frequency domain Fourier coefficients. The transient time interval and the Fourier analysis options are specified on the `.TRAN` and `FOURIER` control lines.

### 1.4. Analysis at Different Temperatures

All input data for SPICE is assumed to have been measured at 27 deg C (300 deg K). The simulation also assumes a nominal temperature of 27 deg C. The circuit can be simulated at other temperatures by using a `.TEMP` control line.

Temperature appears explicitly in the exponential terms of the BJT and diode model equations. In addition, saturation currents have a built-in temperature dependence. The

temperature dependence of the saturation current in the BJT models is determined by:

$$IS(T1) = IS(T0) \left[ \left( \frac{T1}{T0} \right)^{XTI} e^{-\frac{qEG(T1-T0)}{kT1T0}} \right]$$

where  $k$  is Boltzmann's constant,  $q$  is the electronic charge,  $EG$  is the energy gap which is a model parameter, and  $XTI$  is the saturation current temperature exponent (also a model parameter, and usually equal to 3). The temperature dependence of forward and reverse beta is according to the formula:

$$\beta(T1) = \beta(T0) \left[ \left( \frac{T1}{T0} \right)^{XTB} \right]$$

where  $T1$  and  $T0$  are in degrees Kelvin, and  $XTB$  is a user-supplied model parameter. Temperature effects on beta are carried out by appropriate adjustment to the values of  $BF$ ,  $ISE$ ,  $BR$ , and  $ISC$ . Temperature dependence of the saturation current in the junction diode model is determined by:

$$IS(T1) = IS(T0) \left[ \left( \frac{T1}{T0} \right)^{\frac{XTI}{N}} e^{-\frac{qEG(T1-T0)}{kNT1T0}} \right]$$

where  $N$  is the emission coefficient, which is a model parameter, and the other symbols have the same meaning as above. Note that for Schottky barrier diodes, the value of the saturation current temperature exponent,  $XTI$ , is usually 2.

Temperature appears explicitly in the value of junction potential,  $PHI$ , for all the device models. The temperature dependence is determined by:

$$PHI(TEMP) = \frac{kTEMP}{q} \log \left( \frac{NaNd}{Ni(TEMP)^2} \right)$$

where  $k$  is Boltzmann's constant,  $q$  is the electronic charge,  $Na$  is the acceptor impurity density,  $Nd$  is the donor impurity density,  $Ni$  is the intrinsic concentration, and  $EG$  is the energy gap.

Temperature appears explicitly in the value of surface mobility,  $UO$ , for the MOSFET model. The temperature dependence is determined by:

$$UO(TEMP) = \frac{UO(TNOM)}{(TEMP/TNOM)^{1.5}}$$

The effects of temperature on resistors is modeled by the formula:

$$\text{value}(TEMP) = \text{value}(TNOM) [1 + TC1(TEMP - TNOM) + TC2((TEMP - TNOM)^2)]$$

where  $TEMP$  is the circuit temperature,  $TNOM$  is the nominal temperature, and  $TC1$  and  $TC2$  are the first- and second-order temperature coefficients.

## 2. CONVERGENCE

Both dc and transient solutions are obtained by an iterative process which is terminated when both of the following conditions hold:

- 1) The nonlinear branch currents converge to within a tolerance of 0.1 percent or 1 picoamp (1.0E-12 Amp), whichever is larger.
- 2) The node voltages converge to within a tolerance of 0.1 per cent or 1 microvolt (1.0E-6 Volt), whichever is larger.

Although the algorithm used in SPICE has been found to be very reliable, in some cases it will fail to converge to a solution. When this failure occurs, the program will print the node voltages at the last iteration and terminate the job. In such cases, the node voltages that are printed are not necessarily correct or even close to the correct solution.

Failure to converge in the dc analysis is usually due to an error in specifying circuit connections, element values, or model parameter values. Regenerative switching circuits or circuits with positive feedback probably will not converge in the dc analysis unless the **OFF** option is used for some of the devices in the feedback path, or the **NODESET** card is used to

force the circuit to converge to the desired state.

### 3. INPUT FORMAT

The input format for SPICE is of the free format type. Fields on a card are separated by one or more blanks, a comma, an equal (=) sign, or a left or right parenthesis; extra spaces are ignored. A card may be continued by entering a + (plus) in column 1 of the following card; SPICE continues reading beginning with column 2.

A name field must begin with a letter (A through Z) and cannot contain any delimiters. Only the first eight characters of the name are used.

A number field may be an integer field (12, -44), a floating point field (3.14159), either an integer or floating point number followed by an integer exponent (1E-14, 2.65E3), or either an integer or a floating point number followed by one of the following scale factors:

T=1E12 G=1E9 MEG=1E6 K=1E3 MIL=25.4E-6 M=1E-3 U=1E-6 N=1E-9  
P=1E-12 F=1E-15

Letters immediately following a number that are not scale factors are ignored, and letters immediately following a scale factor are ignored. Hence, 10, 10V, 10VOLTS, and 10HZ all represent the same number, and M, MA, MSEC, and MMHOS all represent the same scale factor. Note that 1000, 1000.0, 1000HZ, 1E3, 1.0E3, 1KHZ, and 1K all represent the same number.

### 4. CIRCUIT DESCRIPTION

The circuit to be analyzed is described to SPICE by a set of element cards, which define the circuit topology and element values, and a set of control cards, which define the model parameters and the run controls. The first card in the input deck must be a title card, and the last card must be a .END card. The order of the remaining cards is arbitrary (except, of course, that continuation cards must immediately follow the card being continued).

Each element in the circuit is specified by an element card that contains the element name, the circuit nodes to which the element is connected, and the values of the parameters that determine the electrical characteristics of the element. The first letter of the element name specifies the element type. The format for the SPICE element types is given in what follows. The strings XXXXXXXX, YYYYYYYY, and ZZZZZZZZ denote arbitrary alphanumeric strings. For example, a resistor name must begin with the letter R and can contain from one to eight characters. Hence, R, R1, RSE, ROUT, and R3AC2ZY are valid resistor names.

Data fields that are enclosed in lt and gt signs '< >' are optional. All indicated punctuation (parentheses, equal signs, etc.) are required. With respect to branch voltages and currents, SPICE uniformly uses the associated reference convention (current flows in the direction of voltage drop).

Nodes must be nonnegative integers but need not be numbered sequentially. The datum (ground) node must be numbered zero. The circuit cannot contain a loop of voltage sources and/or inductors and cannot contain a cutset of current sources and/or capacitors. Each node in the circuit must have a dc path to ground. Every node must have at least two connections except for transmission line nodes (to permit unterminated transmission lines) and MOSFET substrate nodes (which have two internal connections anyway).

**5. TITLE CARD, COMMENT CARDS AND .END CARD****5.1. Title Card****Examples:**

**POWER AMPLIFIER CIRCUIT  
TEST OF CAM CELL**

This card must be the first card in the input deck. Its contents are printed verbatim as the heading for each section of output.

**5.2. .END Card****Examples:**

**.END**

This card must always be the last card in the input deck. Note that the period is an integral part of the name.

**5.3. Comment Card****General Form:**

**\* < any comment >**

**Examples:**

**\* RF=1K    GAIN SHOULD BE 100  
\* MAY THE FORCE BE WITH MY CIRCUIT**

The asterisk in the first column indicates that this card is a comment card. Comment cards may be placed anywhere in the circuit description.

**6. ELEMENT CARDS****6.1. Resistors****General form:**

**RXXXXXXXX N1 N2 VALUE < TC=TC1<,TC2> >**

**Examples:**

**R1 1 2 100  
RC1 12 17 1K TC=-0.001,0.015**

N1 and N2 are the two element nodes. VALUE is the resistance (in ohms) and may be positive or negative but not zero. TC1 and TC2 are the (optional) temperature coefficients; if not specified, zero is assumed for both. The value of the resistor as a function of temperature is given by:

$$\text{value (TEMP)} = \text{value (TNOM)} [1 + TC1 (TEMP - TNOM) + TC2 ((TEMP - TNOM)^2)]$$

## 6.2. Capacitors and Inductors

General form:

```
CXXXXXXX N+ N- VALUE <IC=INCOND>
LYYYYYYY N+ N- VALUE <IC=INCOND>
```

Examples:

```
CBYP 13 0 1UF
COSC 17 23 10U IC=3V
LLINK 42 69 1UH
LSHUNT 23 51 10U IC=15.7MA
```

N+ and N- are the positive and negative element nodes, respectively. VALUE is the capacitance in Farads or the inductance in Henries.

For the capacitor, the (optional) initial condition is the initial (time-zero) value of capacitor voltage (in Volts). For the inductor, the (optional) initial condition is the initial (time-zero) value of inductor current (in Amps) that flows from N+, through the inductor, to N-. Note that the initial conditions (if any) apply 'only' if the UIC option is specified on the .TRAN card.

Nonlinear capacitors and inductors can be described.

General form :

```
CXXXXXXX N+ N- POLY C0 C1 C2 ... <IC=INCOND>
LYYYYYYY N+ N- POLY L0 L1 L2 ... <IC=INCOND>
```

C0 C1 C2 ...(and L0 L1 L2 ...) are the coefficients of a polynomial describing the element value. The capacitance is expressed as a function of the voltage across the element while the inductance is a function of the current through the inductor. The value is computed as

$$\text{value} = C0 + C1 \cdot V + C2 \cdot V^2 + \dots$$

$$\text{value} = L0 + L1 \cdot I + L2 \cdot I^2 + \dots$$

where V is the voltage across the capacitor and I the current flowing in the inductor.

## 6.3. Coupled (Mutual) Inductors

General form:

```
KXXXXXXX LYYYYYYY LZZZZZZZ VALUE
```

Examples:

```
K43 LAA LBB 0.999
KXFRMR L1 L2 0.87
```

LYYYYYYY and LZZZZZZZ are the names of the two coupled inductors, and VALUE is the coefficient of coupling, K, which must be greater than 0 and less than or equal to 1. Using the 'dot' convention, place a 'dot' on the first node of each inductor.

#### 6.4. Transmission Lines (Lossless)

General form:

```

TXXXXXXX N1 N2 N3 N4 Z0=VALUE <TD=VALUE> <F=FREQ <NL=NRMLEN> >
+
      <IC=V1,I1,V2,I2>

```

Examples:

```
T1 1 0 2 0 Z0=50 TD=10NS
```

N1 and N2 are the nodes at port 1; N3 and N4 are the nodes at port 2. Z0 is the characteristic impedance. The length of the line may be expressed in either of two forms. The transmission delay, TD, may be specified directly (as TD=10ns, for example). Alternatively, a frequency F may be given, together with NL, the normalized electrical length of the transmission line with respect to the wavelength in the line at the frequency F. If a frequency is specified but NL is omitted, 0.25 is assumed (that is, the frequency is assumed to be the quarter-wave frequency). Note that although both forms for expressing the line length are indicated as optional, one of the two must be specified.

Note that this element models only one propagating mode. If all four nodes are distinct in the actual circuit, then two modes may be excited. To simulate such a situation, two transmission line elements are required. (see the example in Appendix A for further clarification.)

The (optional) initial condition specification consists of the voltage and current at each of the transmission line ports. Note that the initial conditions (if any) apply 'only' if the UIC option is specified on the .TRAN card.

One should be aware that SPICE will use a transient time step which does not exceed 1/2 the minimum transmission line delay. Therefore very short transmission lines (compared with the analysis time frame) will cause long run times.

#### 6.5. Linear Dependent Sources

SPICE allows circuits to contain linear dependent sources characterized by any of the four equations

$$i = g \cdot v \quad v = e \cdot v \quad i = f \cdot i \quad v = h \cdot i$$

where g, e, f, and h are constants representing transconductance, voltage gain, current gain, and transresistance, respectively. Note: a more complete description of dependent sources as implemented in SPICE is given in Appendix B.

#### 6.6. Linear Voltage-Controlled Current Sources

General form:

```
GXXXXXXX N+ N- NC+ NC- VALUE
```

Examples:

```
G1 2 0 5 0 0.1MMHO
```

N+ and N- are the positive and negative nodes, respectively. Current flow is from the positive node, through the source, to the negative node. NC+ and NC- are the positive and negative controlling nodes, respectively. VALUE is the transconductance (in mhos).

**6.7. Linear Voltage-Controlled Voltage Sources****General form:****EXXXXXXX N+ N- NC+ NC- VALUE****Examples:****E1 2 3 14 1 2.0**

N+ is the positive node, and N- is the negative node. NC+ and NC- are the positive and negative controlling nodes, respectively. VALUE is the voltage gain.

**6.8. Linear Current-Controlled Current Sources****General form:****FXXXXXXX N+ N- VNAME VALUE****Examples:****F1 13 5 VSENS 5**

N+ and N- are the positive and negative nodes, respectively. Current flow is from the positive node, through the source, to the negative node. VNAME is the name of a voltage source through which the controlling current flows. The direction of positive controlling current flow is from the positive node, through the source, to the negative node of VNAME. VALUE is the current gain.

**6.9. Linear Current-Controlled Voltage Sources****General form:****HXXXXXXX N+ N- VNAME VALUE****Examples:****HX 5 17 VZ 0.5K**

N+ and N- are the positive and negative nodes, respectively. VNAME is the name of a voltage source through which the controlling current flows. The direction of positive controlling current flow is from the positive node, through the source, to the negative node of VNAME. VALUE is the transresistance (in ohms).

**6.10. Independent Sources****General form:**

**VXXXXXXX N+ N- <<DC> DC/TRAN VALUE> <AC <ACMAG <ACPHASE>>>**  
**IYYYYYYY N+ N- <<DC> DC/TRAN VALUE> <AC <ACMAG <ACPHASE>>>**

**Examples:****VCC 10 0 DC 6**



```
VIN 13 2 0.001 AC 1 SIN(0 1 1MEG)
ISRC 23 21 AC 0.333 45.0 SFFM(0 1 10K 5 1K)
VMEAS 12 9
```

N+ and N- are the positive and negative nodes, respectively. Note that voltage sources need not be grounded. Positive current is assumed to flow from the positive node, through the source, to the negative node. A current source of positive value, will force current to flow out of the N+ node, through the source, and into the N- node. Voltage sources, in addition to being used for circuit excitation, are the 'ammeters' for SPICE, that is, zero valued voltage sources may be inserted into the circuit for the purpose of measuring current. They will, of course, have no effect on circuit operation since they represent short-circuits.

DC/TRAN is the dc and transient analysis value of the source. If the source value is zero both for dc and transient analyses, this value may be omitted. If the source value is time-invariant (e.g., a power supply), then the value may optionally be preceded by the letters DC.

ACMAG is the ac magnitude and ACPHASE is the ac phase. The source is set to this value in the ac analysis. If ACMAG is omitted following the keyword AC, a value of unity is assumed. If ACPHASE is omitted, a value of zero is assumed. If the source is not an ac small-signal input, the keyword AC and the ac values are omitted.

Any independent source can be assigned a time-dependent value for transient analysis. If a source is assigned a time-dependent value, the time-zero value is used for dc analysis. There are five independent source functions: pulse, exponential, sinusoidal, piecewise linear, and single-frequency FM. If parameters other than source values are omitted or set to zero, the default values shown will be assumed. (TSTEP is the printing increment and TSTOP is the final time (see the .TRAN card for explanation)).

1. Pulse      PULSE(V1 V2 TD TR TF PW PER)

Examples:

```
VIN 3 0 PULSE(-1 1 2NS 2NS 2NS 50NS 100NS)
```

parameter	default	units
V1 (initial value)		Volts or Amps
V2 (pulsed value)		Volts or Amps
TD (delay time)	0.0	seconds
TR (rise time)	TSTEP	seconds
TF (fall time)	TSTEP	seconds
PW (pulse width)	TSTOP	seconds
PER(period)	TSTOP	seconds

A single pulse so specified is described by the following table:

time	value
0	V1
TD	V1
TD+TR	V2
TD+TR+PW	V2
TD+TR+PW+TF	V1
TSTOP	V1

Intermediate points are determined by linear interpolation.

## 2. Sinusoidal SIN(VO VA FREQ TD THETA)

Examples:

```
VIN 3 0 SIN(0 1 100MEG 1NS 1E10)
```

parameter	default value	units
VO	(offset)	Volts or Amps
VA	(amplitude)	Volts or Amps
FREQ	(frequency)	1/TSTOP
TD	(delay)	0.0
THETA	(damping factor)	0.0

The shape of the waveform is described by the following table:

time	value
0 to TD	VO
TD to TSTOP	$VO + VA e^{-(\text{time} - TD)\Theta} \sin(2\pi \text{FREQ}(\text{time} + TD))$

## 3. Exponential EXP(V1 V2 TD1 TAU1 TD2 TAU2)

Examples:

```
VIN 3 0 EXP(-4 -1 2NS 30NS 60NS 40NS)
```

parameters	default values	units
V1	(initial value)	Volts or Amps
V2	(pulsed value)	Volts or Amps
TD1	(rise delay time)	0.0
TAU1	(rise time constant)	TSTEP
TD2	(fall delay time)	TD1+TSTEP
TAU2	(fall time constant)	TSTEP

The shape of the waveform is described by the following table:

time	value
0 to TD1	V1
TD1 to TD2	$V1 + (V2 - V1)[1 - e^{-(\text{time} - TD1)/TAU1}]$
TD2 to TSTOP	$V1 + (V2 - V1)[1 - e^{-(\text{time} - TD1)/TAU1}] + (V1 - V2)[1 - e^{-(\text{time} - TD2)/TAU2}]$

## 4. Piece-Wise Linear PWL(T1 V1 < T2 V2 T3 V3 T4 V4 ...)

Example:

VCLOCK 7 5 PWL(0 -7 10NS -7 11NS -3 17NS -3 18NS -7 50NS -7)

Each pair of values (Ti, Vi) specifies that the value of the source is Vi (in Volts or Amps) at time=Ti. The value of the source at intermediate values of time is determined by using linear interpolation on the input values.

### 5. Single-Frequency FM SFFM(VO VA FC MDI FS)

Examples:

V1 12 0 SFFM(0 1M 20K 5 1K)

parameters	default values	units
VO	(offset)	Volts or Amps
VA	(amplitude)	Volts or Amps
FC	(carrier frequency)	1/TSTOP
MDI	(modulation index)	
FS	(signal frequency)	1/TSTOP

The shape of the waveform is described by the following equation:

$$\text{value} + \text{VO} + \text{VA} \sin((2\pi \cdot \text{FC} \cdot \text{time}) + \text{MDI} \sin(2\pi \cdot \text{FS} \cdot \text{time}))$$

## 7. SEMICONDUCTOR DEVICES

The elements that have been described to this point typically require only a few parameter values to specify completely the electrical characteristics of the element. However, the models for the four semiconductor devices that are included in the SPICE program require many parameter values. Moreover, many devices in a circuit often are defined by the same set of device model parameters. For these reasons, a set of device model parameters is defined on a separate MODEL card and assigned a unique model name. The device element cards in SPICE then reference the model name. This scheme alleviates the need to specify all of the model parameters on each device element card.

Each device element card contains the device name, the nodes to which the device is connected, and the device model name. In addition, other optional parameters may be specified for each device: geometric factors and an initial condition.

The area factor used on the diode, BJT and JFET device card determines the number of equivalent parallel devices of a specified model. The affected parameters are marked with an asterisk under the heading 'area' in the model descriptions below. Several geometric factors associated with the channel and the drain and source diffusions can be specified on the MOSFET device card.

Two different forms of initial conditions may be specified for devices. The first form is included to improve the dc convergence for circuits that contain more than one stable state. If a device is specified OFF, the dc operating point is determined with the terminal voltages for that device set to zero. After convergence is obtained, the program continues to iterate to obtain the exact value for the terminal voltages. If a circuit has more than one dc stable state, the OFF option can be used to force the solution to correspond to a desired state. If a device is specified OFF when in reality the device is conducting, the program will still obtain the correct solution (assuming the solutions converge) but more iterations will be required since the program must independently converge to two separate solutions. The .NODESET card serves a similar purpose as the OFF option. The .NODESET

option is easier to apply and is the preferred means to aid convergence.

The second form of initial conditions are specified for use with the transient analysis. These are true 'initial conditions' as opposed to the convergence aids above. See the description of the .IC card and the .TRAN card for a detailed explanation of initial conditions.

### 7.1. Junction Diodes

General form:

**DXXXXXXX N+ N- MNAME <AREA> <OFF> <IC=VD>**

Examples:

**DBRIDGE 2 10 DIODE1  
DCLMP 3 7 DMOD 3.0 IC=0.2**

N+ and N- are the positive and negative nodes, respectively. MNAME is the model name, AREA is the area factor, and off indicates an (optional) starting condition on the device for dc analysis. If the area factor is omitted, a value of 1.0 is assumed. The (optional) initial condition specification using IC=VD is intended for use with the UIC option on the .TRAN card, when a transient analysis is desired starting from other than the quiescent operating point.

### 7.2. Bipolar Junction Transistors (BJT's)

General form:

**QXXXXXXX NC NB NE <NS> MNAME <AREA> <OFF> <IC=VBE,VCE>**

Examples:

**Q23 10 24 13 QMOD IC=0.6,5.0  
Q50A 11 26 4 20 MOD1**

NC, NB, and NE are the collector, base, and emitter nodes, respectively. NS is the (optional) substrate node. If unspecified, ground is used. MNAME is the model name, AREA is the area factor, and OFF indicates an (optional) initial condition on the device for the dc analysis. If the area factor is omitted, a value of 1.0 is assumed. The (optional) initial condition specification using IC=VBE,VCE is intended for use with the UIC option on the .TRAN card, when a transient analysis is desired starting from other than the quiescent operating point. See the .IC card description for a better way to set transient initial conditions.

### 7.3. Junction Field-Effect Transistors (JFET's)

General form:

**JXXXXXXX ND NG NS MNAME <AREA> <OFF> <IC=VDS,VGS>**

Examples:

**J1 7 2 3 JM1 OFF**

ND, NG, and NS are the drain, gate, and source nodes, respectively. MNAME is the model name, AREA is the area factor, and OFF indicates an (optional) initial condition on the device for dc analysis. If the area factor is omitted, a value of 1.0 is assumed. The (optional) initial condition specification, using IC=VDS,VGS is intended for use with the UIC option on the .TRAN card, when a transient analysis is desired starting from other than the quiescent operating point (see the .IC card for a better way to set initial conditions).

#### 7.4. MOSFET's

General form:

```

MXXXXXXX ND NG NS NB MNAME <L=VAL> <W=VAL> <AD=VAL>
<AS=VAL>
+ <PD=VAL> <PS=VAL> <NRD=VAL> <NRS=VAL> <OFF>
<IC=VDS,VGS,VBS>

```

Examples:

```

M1 24 2 0 20 TYPE1
M31 2 17 6 10 MODM L=5U W=2U
M31 2 16 6 10 MODM 5U 2U
M1 2 9 3 0 MOD1 L=10U W=5U AD=100P AS=100P PD=40U PS=40U
M1 2 9 3 0 MOD1 10U 5U 2P 2P

```

ND, NG, NS, and NB are the drain, gate, source, and bulk (substrate) nodes, respectively. MNAME is the model name. L and W are the channel length and width, in meters. AD and AS are the areas of the drain and source diffusions, in sq-meters. Note that the suffix U specifies microns (1E-6 m) and P sq-microns (1E-12 sq-m). If any of L, W, AD, or AS are not specified, default values are used. The user may specify the values to be used for these default parameters on the .OPTIONS card. The use of defaults simplifies input deck preparation, as well as the editing required if device geometries are to be changed. PD and PS are the perimeters of the drain and source junctions, in meters. NRD and NRS designate the equivalent number of squares of the drain and source diffusions; these values multiply the sheet resistance RSH specified on the .MODEL card for an accurate representation of the parasitic series drain and source resistance of each transistor. PD and PS default to 0.0 while NRD and NRS to 1.0. OFF indicates an (optional) initial condition on the device for dc analysis. The (optional) initial condition specification using IC=VDS,VGS,VBS is intended for use with the UIC option on the .TRAN card, when a transient analysis is desired starting from other than the quiescent operating point. See the .IC card for a better and more convenient way to specify transient initial conditions.

#### 7.5. .MODEL Card

General form:

```
.MODEL MNAME TYPE(PNAME1=PVAL1 PNAME2=PVAL2 ... )
```

Examples:

```
.MODEL MOD1 NPN BF=50 IS=1E-13 VBF=50
```

The .MODEL card specifies a set of model parameters that will be used by one or more devices. MNAME is the model name, and type is one of the following seven types:

type	description
------	-------------

NPN	NPN BJT model
PNP	PNP BJT model
D	diode model
NJF	N-channel JFET model
PJF	P-channel JFET model
NMOS	N-channel MOSFET model
PMOS	P-channel MOSFET model

Parameter values are defined by appending the parameter name, as given below for each model type, followed by an equal sign and the parameter value. Model parameters that are not given a value are assigned the default values given below for each model type.

#### 7.6. Diode Model

The dc characteristics of the diode are determined by the parameters IS and N. An ohmic resistance, RS, is included. Charge storage effects are modeled by a transit time, TT, and a nonlinear depletion layer capacitance which is determined by the parameters CJO, VJ, and M. The temperature dependence of the saturation current is defined by the parameters EG, the energy and XTI, the saturation current temperature exponent. Reverse breakdown is modeled by an exponential increase in the reverse diode current and is determined by the parameters BV and IBV (both of which are positive numbers).

	name	parameter	units	default	example	area
1	IS	saturation current	A	1.0E-14	1.0E-14	•
2	RS	ohmic resistance	Ohm	0	10	•
3	N	emission coefficient	-	1	1.0	
4	TT	transit-time	sec	0	0.1Ns	
5	CJO	zero-bias junction capacitance	F	0	2PF	•
6	VJ	junction potential	V	1	0.6	
7	M	grading coefficient	-	0.5	0.5	
8	EG	activation energy	eV	1.11	1.11 Si 0.69 Sbd 0.67 Ge	
9	XTI	saturation-current temp. exp	-	3.0	3.0 jn 2.0 Sbd	
10	KF	flicker noise coefficient	-	0		
11	AF	flicker noise exponent	-	1		
12	FC	coefficient for forward-bias depletion capacitance formula	-	0.5		
13	BV	reverse breakdown voltage	V	infinite	40.0	
14	IBV	current at breakdown voltage	A	1.0E-3		

#### 7.7. BJT Models (both NPN and PNP)

The bipolar junction transistor model in SPICE is an adaptation of the integral charge control model of Gummel and Poon. This modified Gummel-Poon model extends the original model to include several effects at high bias levels. The model will automatically simplify to the simpler Ebers-Moll model when certain parameters are not specified. The parameter names used in the modified Gummel-Poon model have been chosen to be more easily understood by the program user, and to reflect better both physical and circuit design thinking.

The dc model is defined by the parameters IS, BF, NF, ISE, IKF, and NE which determine the forward current gain characteristics, IS, BR, NR, ISC, IKR, and NC which determine the reverse current gain characteristics, and VAF and VAR which determine the

output conductance for forward and reverse regions. Three ohmic resistances RB, RC, and RE are included, where RB can be high current dependent. Base charge storage is modeled by forward and reverse transit times, TF and TR, the forward transit time TF being bias dependent if desired, and nonlinear depletion layer capacitances which are determined by CJE, VJE, and MJE for the B-E junction, CJC, VJC, and MJC for the B-C junction and CJS, VJS, and MJS for the C-S (Collector-Substrate) junction. The temperature dependence of the saturation current, IS, is determined by the energy-gap, EG, and the saturation current temperature exponent, XTI. Additionally base current temperature dependence is modeled by the beta temperature exponent XTB in the new model.

The BJT parameters used in the modified Gummel-Poon model are listed below. The parameter names used in earlier versions of SPICE2 are still accepted.

#### Modified Gummel-Poon BJT Parameters

	name	parameter	units	default	example	area
1	IS	transport saturation current	A	1.0E-16	1.0E-15	*
2	BF	ideal maximum forward beta	-	100	100	
3	NF	forward current emission coefficient	-	1.0	1	
4	VA	forward Early voltage	V	infinite	200	
5	IKF	corner for forward beta				
		high current roll-off	A	infinite	0.01	*
6	ISE	B-E leakage saturation current	A	0	1.0E-13	*
7	NE	B-E leakage emission coefficient	-	1.5	2	
8	BR	ideal maximum reverse beta	-	1	0.1	
9	NR	reverse current emission coefficient	-	1	1	
10	VAR	reverse Early voltage	V	infinite	200	
11	IKR	corner for reverse beta				
		high current roll-off	A	infinite	0.01	*
12	ISC	B-C leakage saturation current	A	0	1.0E-13	*
13	NC	B-C leakage emission coefficient	-	2	1.5	
14	RB	zero bias base resistance	Ohms	0	100	*
15	IRB	current where base resistance				
		falls halfway to its min value	A	infinite	0.1	*
16	RBM	minimum base resistance				
		at high currents	Ohms	RB	10	*
17	RE	emitter resistance	Ohms	0	1	*
18	RC	collector resistance	Ohms	0	10	*
19	CJE	B-E zero-bias depletion capacitance	F	0	2PF	*
20	VJE	B-E built-in potential	V	0.75	0.6	
21	MJE	B-E junction exponential factor	-	0.33	0.33	
22	TF	ideal forward transit time	sec	0	0.1Ns	
23	XTF	coefficient for bias dependence of TF	-	0		
24	VTF	voltage describing VBC				
		dependence of TF	V	infinite		
25	ITF	high-current parameter				
		for effect on TF	A	0	*	
26	PTF	excess phase at freq = 1.0/(TF*2PI)	Hz	deg	0	
27	CJC	B-C zero-bias depletion capacitance	F	0	2PF	*
28	VJC	B-C built-in potential	V	0.75	0.5	
29	MJC	B-C junction exponential factor	-	0.33	0.5	
30	XCJC	fraction of B-C depletion capacitance				
		connected to internal base node	-	1		
31	TR	ideal reverse transit time	sec	0	10Ns	
32	CJS	zero-bias collector-substrate				

		capacitance	F	0	2PF	*
33	VJS	substrate junction built-in potential	V	0.75		
34	MJS	substrate junction exponential factor	-	0	0.5	
35	XTB	forward and reverse beta				
		temperature exponent	-	0		
36	EG	energy gap for temperature effect on IS	eV	1.11		
37	XTI	temperature exponent for effect on IS	-	3		
38	KF	flicker-noise coefficient	-	0		
39	AF	flicker-noise exponent	-	1		
40	FC	coefficient for forward-bias depletion capacitance formula	-	0.5		

### 7.8. JFET Models (both N and P Channel)

The JFET model is derived from the FET model of Shichman and Hodges. The dc characteristics are defined by the parameters VTO and BETA, which determine the variation of drain current with gate voltage, LAMBDA, which determines the output conductance, and IS, the saturation current of the two gate junctions. Two ohmic resistances, RD and RS, are included. Charge storage is modeled by nonlinear depletion layer capacitances for both gate junctions which vary as the  $-1/2$  power of junction voltage and are defined by the parameters CGS, CGD, and PB.

	name	parameter	units	default	example	area
1	VTO	threshold voltage	V	-2.0	-2.0	*
2	BETA	transconductance parameter	A/V <sup>2</sup>	1.0E-4	1.0E-3	*
3	LAMBDA	channel length modulation parameter	1/V	0	1.0E-4	*
4	RD	drain ohmic resistance	Ohm	0	100	*
5	RS	source ohmic resistance	Ohm	0	100	*
6	CGS	zero-bias G-S junction capacitance	F	0	5PF	*
7	CGD	zero-bias G-D junction capacitance	F	0	1PF	*
8	PB	gate junction potential	V	1	0.6	*
9	IS	gate junction saturation current	A	1.0E-14	1.0E-14	*
10	KF	flicker noise coefficient	-	0		
11	AF	flicker noise exponent	-	1		
12	FC	coefficient for forward-bias depletion capacitance formula	-	0.5		

### 7.9. MOSFET Models (both N and P channel)

SPICE provides three MOSFET device models which differ in the formulation of the I-V characteristic. The variable LEVEL specifies the model to be used:

LEVEL=1 -> Shichman-Hodges

LEVEL=2 -> MOS2 (as described in [1])

LEVEL=3 -> MOS3, a semi-empirical model(see [1])

The dc characteristics of the MOSFET are defined by the device parameters VTO, KP, LAMBDA, PHI and GAMMA. These parameters are computed by SPICE if process parameters (NSUB, TOX, ...) are given, but user-specified values always override. VTO is positive (negative) for enhancement mode and negative (positive) for depletion mode N-



channel (P-channel) devices. Charge storage is modeled by three constant capacitors, CGSO, CGDO, and CGBO which represent overlap capacitances, by the nonlinear thin-oxide capacitance which is distributed among the gate, source, drain, and bulk regions, and by the nonlinear depletion-layer capacitances for both substrate junctions divided into bottom and periphery, which vary as the MJ and MJSW power of junction voltage respectively, and are determined by the parameters CBD, CBS, CJ, CJSW, MJ, MJSW and PB. There are two built-in models of the charge storage effects associated with the thin-oxide. The default is the piecewise linear voltage-dependent capacitance model proposed by Meyer. The second choice is the charge-controlled capacitance model of Ward and Dutton [1]. The XQC model parameter acts as a flag and a coefficient at the same time. As the former it causes the program to use Meyer's model whenever larger than 0.5 or not specified, and the charge-controlled model when between 0 and 0.5. In the latter case its value defines the share of the channel charge associated with the drain terminal in the saturation region. The thin-oxide charge storage effects are treated slightly different for the LEVEL=1 model. These voltage-dependent capacitances are included only if TOX is specified in the input description and they are represented using Meyer's formulation.

There is some overlap among the parameters describing the junctions, e.g. the reverse current can be input either as IS (in A) or as JS (in  $A/m^2$ ). Whereas the first is an absolute value the second is multiplied by AD and AS to give the reverse current of the drain and source junctions respectively. This methodology has been chosen since there is no sense in relating always junction characteristics with AD and AS entered on the device card; the areas can be defaulted. The same idea applies also to the zero-bias junction capacitances CBD and CBS (in F) on one hand, and CJ (in  $F/m^2$ ) on the other. The parasitic drain and source series resistance can be expressed as either RD and RS (in ohms) or RSH (in ohms/sq.), the latter being multiplied by the number of squares NRD and NRS input on the device card.

	name	parameter	units	default	example
1	LEVEL	model index	-	1	
2	VTO	zero-bias threshold voltage	V	0.0	1.0
3	KP	transconductance parameter	$A/V^2$	2.0E-5	3.1E-5
4	GAMMA	bulk threshold parameter	$V^{0.5}$	0.0	0.37
5	PHI	surface potential	V	0.6	0.65
6	LAMBDA	channel-length modulation (MOS1 and MOS2 only)	1/V	0.0	0.02
7	RD	drain ohmic resistance	Ohm	0.0	1.0
8	RS	source ohmic resistance	Ohm	0.0	1.0
9	CBD	zero-bias B-D junction capacitance	F	0.0	20FF
10	CBS	zero-bias B-S junction capacitance	F	0.0	20FF
11	IS	bulk junction saturation current	A	1.0E-14	1.0E-15
12	PB	bulk junction potential	V	0.8	0.87
13	CGSO	gate-source overlap capacitance per meter channel width	F/m	0.0	4.0E-11
14	CGDO	gate-drain overlap capacitance per meter channel width	F/m	0.0	4.0E-11
15	CGBO	gate-bulk overlap capacitance per meter channel length	F/m	0.0	2.0E-10
16	RSH	drain and source diffusion sheet resistance	Ohm/sq.	0.0	10.0

[1] A. Vladimirescu and S. Liu, "The Simulation of MOS Integrated Circuits Using SPICE2", ERL Memo No. ERL M80/7, Electronics Research Laboratory, University of California, Berkeley, Oct. 1980.

17	CJ	zero-bias bulk junction bottom cap. per sq-meter of junction area	$F/m^2$	0.0	2.0E-4
18	MJ	bulk junction bottom grading coef.	-	0.5	0.5
19	CJSW	zero-bias bulk junction sidewall cap. per meter of junction perimeter	$F/m$	0.0	1.0E-9
20	MJSW	bulk junction sidewall grading coef.	-	0.33	
21	JS	bulk junction saturation current per sq-meter of junction area	$A/m^2$	1.0E-8	
22	TOX	oxide thickness	meter	1.0E-7	1.0E-7
23	NSUB	substrate doping	$1/cm^3$	0.0	4.0E15
24	NSS	surface state density	$1/cm^2$	0.0	1.0E10
25	NFS	fast surface state density	$1/cm^2$	0.0	1.0E10
26	TPG	type of gate material: +1 opp. to substrate -1 same as substrate 0 Al gate	-	1.0	
27	XJ	metallurgical junction depth	meter	0.0	1U
28	LD	lateral diffusion	meter	0.0	0.8U
29	UO	surface mobility	$cm^2/V-s$	600	700
30	UCRIT	critical field for mobility degradation (MOS2 only)	$V/cm$	1.0E4	1.0E4
31	UEXP	critical field exponent in mobility degradation (MOS2 only)	-	0.0	0.1
32	UTRA	transverse field coeff (mobility) (deleted for MOS2)	-	0.0	0.3
33	VMAX	maximum drift velocity of carriers	$m/s$	0.0	5.0E4
34	NEFF	total channel charge (fixed and mobile) coefficient (MOS2 only)	-	1.0	5.0
35	XQC	thin-oxide capacitance model flag and coefficient of channel charge share attributed to drain (0-0.5)	-	1.0	0.4
36	KF	flicker noise coefficient	-	0.0	1.0E-26
37	AF	flicker noise exponent	-	1.0	12
38	FC	coefficient for forward-bias depletion capacitance formula	-	0.5	
39	DELTA	width effect on threshold voltage (MOS2 and MOS3)	-	0.0	1.0
40	THETA	mobility modulation (MOS3 only)	$1/V$	0.0	0.1
41	ETA	static feedback (MOS3 only)	-	0.0	1.0
42	KAPPA	saturation field factor (MOS3 only)	-	0.2	0.5

## 8. SUBCIRCUITS

A subcircuit that consists of SPICE elements can be defined and referenced in a fashion similar to device models. The sub-circuit is defined in the input deck by a grouping of element cards; the program then automatically inserts the group of elements wherever the subcircuit is referenced. There is no limit on the size or complexity of subcircuits, and subcircuits may contain other subcircuits. An example of subcircuit usage is given in Appendix A.

### 8.1. .SUBCKT Card

General form:

**.SUBCKT** subname N1 < N2 N3 ...>

**Examples:**

**.SUBCKT OPAMP 1 2 3 4**

A circuit definition is begun with a **.SUBCKT** card. **SUBNAM** is the subcircuit name, and **N1, N2, ...** are the external nodes, which cannot be zero. The group of element cards which immediately follow the **.SUBCKT** card define the subcircuit. The last card in a subcircuit definition is the **.ENDS** card (see below). Control cards may not appear within a subcircuit definition; however, subcircuit definitions may contain anything else, including other subcircuit definitions, device models, and subcircuit calls (see below). Note that any device models or subcircuit definitions included as part of a subcircuit definition are strictly local (i.e., such models and definitions are not known outside the subcircuit definition). Also, any element nodes not included on the **.SUBCKT** card are strictly local, with the exception of 0 (ground) which is always global.

**8.2. .ENDS Card****General form:**

**.ENDS <SUBNAM>**

**Examples:**

**.ENDS OPAMP**

This card must be the last one for any subcircuit definition. The subcircuit name, if included, indicates which subcircuit definition is being terminated; if omitted, all subcircuits being defined are terminated. The name is needed only when nested subcircuit definitions are being made.

**8.3. Subcircuit Calls****General form:**

**XXXXXXXXY N1 <N2 N3 ...> SUBNAM**

**Examples:**

**X1 2 4 17 3 1 MULTI**

Subcircuits are used in SPICE by specifying pseudo-elements beginning with the letter **X**, followed by the circuit nodes to be used in expanding the subcircuit.

**9. CONTROL CARDS****9.1. .TEMP Card****General form:**

**.TEMP T1 <T2 <T3 ...> >**

**Examples:**

**.TEMP -55.0 25.0 125.0**

This card specifies the temperatures at which the circuit is to be simulated. T1, T2, ... Are the different temperatures, in degrees C. Temperatures less than -223.0 deg C are ignored. Model data are specified at TNOM degrees (see the .OPTIONS card for TNOM); if the .TEMP card is omitted, the simulation will also be performed at a temperature equal to TNOM.

### 9.2. .WIDTH Card

General form:

**.WIDTH IN=COLNUM OUT=COLNUM**

Examples:

**.WIDTH IN=72 OUT=133**

COLNUM is the last column read from each line of input; the setting takes effect with the next line read. The default value for COLNUM is 80. The out parameter specifies the output print width. Permissible values for the output print width are 80 and 133.

### 9.3. .OPTIONS Card

General form:

**.OPTIONS OPT1 OPT2 ... (or OPT=OPTVAL ...)**

Examples:

**.OPTIONS ACCT LIST NODE**

This card allows the user to reset program control and user options for specific simulation purposes. Any combination of the following options may be included, in any order. 'x' (below) represents some positive number.

option	effect
ACCT	causes accounting and run time statistics to be printed
LIST	causes the summary listing of the input data to be printed
NOMOD	suppresses the printout of the model parameters.
NOPAGE	suppresses page ejects
NODE	causes the printing of the node table.
OPTS	causes the option values to be printed.
GMIN=x	sets the value of GMIN, the minimum conductance allowed by the program. The default value is 1.0E-12.
RELTOL=x	resets the relative error tolerance of the program. The default value is 0.001 (0.1 percent).
ABSTOL=x	resets the absolute current error tolerance of the program. The default value is 1 picoamp.
VNTOL=x	resets the absolute voltage error tolerance of the program. The default value is 1 microvolt.

<b>TRTOL=x</b>	resets the transient error tolerance. The default value is 7.0. This parameter is an estimate of the factor by which SPICE overestimates the actual truncation error. IP "CHGTOL=x" 17 resets the charge tolerance of the program. The default value is 1.0E-14.
<b>PIVTOL=x</b>	resets the absolute minimum value for a matrix entry to be accepted as a pivot. The default value is 1.0E-13.
<b>PIVREL=x</b>	resets the relative ratio between the largest column entry and an acceptable pivot value. The default value is 1.0E-3. In the numerical pivoting algorithm the allowed minimum pivot value is determined by $EPSREL = AMAX1(PIVREL \cdot MAXVAL, PIVTOL)$ where MAXVAL is the maximum element in the column where a pivot is sought (partial pivoting).
<b>NUMDGT=x</b>	is the number of significant digits printed for output variable values. X must satisfy the relation $0 < x < 8$ . The default value is 4. Note: this option is independent of the error tolerance used by SPICE (i.e., if the values of options RELTOL, ABSTOL, etc., are not changed then one may be printing numerical 'noise' for NUMDGT > 4.
<b>TNOM=x</b>	resets the nominal temperature. The default value is 27 deg C (300 deg K).
<b>ITL1=x</b>	resets the dc iteration limit. The default is 100.
<b>ITL2=x</b>	resets the dc transfer curve iteration limit. The default is 50.
<b>ITL3=x</b>	resets the lower transient analysis iteration limit. The default value is 4.
<b>ITL4=x</b>	resets the transient analysis timepoint iteration limit. The default is 10.
<b>ITL5=x</b>	resets the transient analysis total iteration limit. The default is 5000. Set ITL5=0 to omit this test.
<b>ITL6=x</b>	resets the dc iteration limit at each step of the source stepping method. The default is 0 which means not to use this method.
<b>CPTIME=x</b>	is the maximum cpu-time in seconds allowed for this job.
<b>LIMTIM=x</b>	resets the amount of cpu time reserved by SPICE for generating plots should a cpu time-limit cause job termination. The default value is 2 (seconds).
<b>LIMPTS=x</b>	resets the total number of points that can be printed or plotted in a dc, ac, or transient analysis. The default value is 201.
<b>LVLCOD=x</b>	if x is 2 (two), then machine code for the matrix solution will be generated. Otherwise, no machine code is generated. The default value is 2. Applies only to CDC computers.
<b>LVLTIM=x</b>	is 1 (one), the iteration timestep control is used. if x is 2 (two), the truncation-error timestep is used. The default value is 2. If method=Gear and MAXORD > 2 then LVLTIM is set to 2 by SPICE.
<b>METHOD=name</b>	sets the numerical integration method used by SPICE. Possible names are Gear or trapezoidal. The default is trapezoidal.
<b>MAXORD=x</b>	sets the maximum order for the integration method if Gear's variable-order method is used. X must be between 2 and 6. The default value is 2.
<b>DEFL=x</b>	resets the value for MOS channel length; the default is 100.0 micrometer.
<b>DEFW=x</b>	resets the value for MOS channel width; the default is 100.0 micrometer.
<b>DEFAD=x</b>	resets the value for MOS drain diffusion area; the default is 0.0.

**DEFAS=x** resets the value for MOS source diffusion area; the default is 0.0.

#### 9.4. .OP Card

**General form:**

**.OP**

The inclusion of this card in an input deck will force SPICE to determine the dc operating point of the circuit with inductors shorted and capacitors opened. Note: a dc analysis is automatically performed prior to a transient analysis to determine the transient initial conditions, and prior to an ac small-signal analysis to determine the linearized, small-signal models for nonlinear devices.

SPICE performs a dc operating point analysis if no other analyses are requested.

#### 9.5. .DC Card

**General form:**

**.DC SRCNAM VSTART VSTOP VINC [SRC2 START2 STOP2 INCR2]**

**Examples:**

```
.DC VIN 0.25 5.0 0.25
.DC VDS 0 10 .5 VGS 0 5 1
.DC VCE 0 10 .25 IB 0 10U 1U
```

This card defines the dc transfer curve source and sweep limits. SRCNAM is the name of an independent voltage or current source. VSTART, VSTOP, and VINC are the starting, final, and incrementing values respectively. The first example will cause the value of the voltage source VIN to be swept from 0.25 Volts to 5.0 Volts in increments of 0.25 Volts. A second source (SRC2) may optionally be specified with associated sweep parameters. In this case, the first source will be swept over its range for each value of the second source. This option can be useful for obtaining semiconductor device output characteristics. See the second example data deck in that section of the guide.

#### 9.6. .NODESET Card

**General form:**

**.NODESET V(NODNUM)=VAL V(NODNUM)=VAL ...**

**Examples:**

```
.NODESET V(12)=4.5 V(4)=2.23
```

This card helps the program find the dc or initial transient solution by making a preliminary pass with the specified nodes held to the given voltages. The restriction is then released and the iteration continues to the true solution. The NODESET card may be necessary for convergence on bistable or astable circuits. In general, this card should not be necessary.

### 9.7. IC Card

General form:

**.IC V(NODNUM)=VAL V(NODNUM)=VAL ...**

Examples:

**.IC V(11)=5 V(4)=-5 V(2)=2.2**

This card is for setting transient initial conditions. It has two different interpretations, depending on whether the UIC parameter is specified on the .TRAN card. Also, one should not confuse this card with the NODESET card. The NODESET card is only to help dc convergence, and does not affect final bias solution (except for multi-stable circuits). The two interpretations of this card are as follows:

1. When the UIC parameter is specified on the .TRAN card, then the node voltages specified on the IC card are used to compute the capacitor, diode, BJT, JFET, and MOSFET initial conditions. This is equivalent to specifying the IC=... parameter on each device card, but is much more convenient. The IC=... parameter can still be specified and will take precedence over the IC values. Since no dc bias (initial transient) solution is computed before the transient analysis, one should take care to specify all dc source voltages on the IC card if they are to be used to compute device initial conditions.
2. When the UIC parameter is not specified on the .TRAN card, the dc bias (initial transient) solution will be computed before the transient analysis. In this case, the node voltages specified on the IC card will be forced to the desired initial values during the bias solution. During transient analysis, the constraint on these node voltages is removed.

### 9.8. TF Card

General form:

**.TF OUTVAR INSRC**

Examples:

**.TF V(5,3) VIN**

**.TF I(VLOAD) VIN**

This card defines the small signal output and input for the dc small signal analysis. OUTVAR is the small-signal output variable and INSRC is the small-signal input source. If this card is included, SPICE will compute the dc small signal value of the transfer function (output/input), input resistance, and output resistance. For the first example, SPICE would compute the ratio of V(5,3) to VIN, the small-signal input resistance at VIN, and the small-signal output resistance measured across nodes 5 and 3.

### 9.9. SENS Card

General form:

**.SENS OV1 <OV2 ... >**

**Examples:**

```
.SENS V(9) V(4,3) V(17) I(VCC)
```

If a .SENS card is included in the input deck, SPICE will determine the dc small-signal sensitivities of each specified output variable with respect to every circuit parameter. Note: for large circuits, large amounts of output can be generated.

**9.10. .AC Card****General form:**

```
.AC DEC ND FSTART FSTOP
.AC OCT NO FSTART FSTOP
.AC LIN NP FSTART FSTOP
```

**Examples:**

```
.AC DEC 10 1 10K
.AC DEC 10 1K 100MEG
.AC LIN 100 1 100HZ
```

DEC stands for decade variation, and ND is the number of points per decade. OCT stands for octave variation, and NO is the number of points per octave. LIN stands for linear variation, and NP is the number of points. FSTART is the starting frequency, and FSTOP is the final frequency. If this card is included in the deck, SPICE will perform an ac analysis of the circuit over the specified frequency range. Note that in order for this analysis to be meaningful, at least one independent source must have been specified with an ac value.

**9.11. .DISTO Card****General form:**

```
.DISTO RLOAD <INTER <SKW2 <REFFWR <SPW2> > > >
```

**Examples:**

```
.DISTO RL 2 0.95 1.0E-3 0.75
```

This card controls whether SPICE will compute the distortion characteristic of the circuit in a small-signal mode as a part of the ac small-signal sinusoidal steady-state analysis. The analysis is performed assuming that one or two signal frequencies are imposed at the input; let the two frequencies be f1 (the nominal analysis frequency) and f2 ( $=SKW2 \cdot f1$ ). The program then computes the following distortion measures:

- HD2** - the magnitude of the frequency component  $2 \cdot f1$  assuming that f2 is not present.
- HD3** - the magnitude of the frequency component  $3 \cdot f1$  assuming that f2 is not present.
- SIM2** - the magnitude of the frequency component  $f1 + f2$ .
- DIM2** - the magnitude of the frequency component  $f1 - f2$ .



**DIM3** - the magnitude of the frequency component  $2 \cdot f_1 - f_2$ .

**RLOAD** is the name of the output load resistor into which all distortion power products are to be computed. **INTER** is the interval at which the summary printout of the contributions of all nonlinear devices to the total distortion is to be printed. If omitted or set to zero, no summary printout will be made. **REFPWR** is the reference power level used in computing the distortion products; if omitted, a value of 1 mW (that is, dbm) is used. **SKW2** is the ratio of  $f_2$  to  $f_1$ . If omitted, a value of 0.9 is used (i.e.,  $f_2 = 0.9 \cdot f_1$ ). **SPW2** is the amplitude of  $f_2$ . If omitted, a value of 1.0 is assumed. The distortion measures **HD2**, **HD3**, **SIM2**, **DIM2**, and **DIM3** may also be printed and/or plotted (see the description of the **PRINT** and **PLOT** cards).

### 9.12. **.NOISE Card**

General form:

**.NOISE OUTV INSRC NUMS**

Examples:

**.NOISE V(5) VIN 10**

This card controls the noise analysis of the circuit. The noise analysis is performed in conjunction with the ac analysis (see **.AC card**). **OUTV** is an output voltage which defines the summing point. **INSRC** is the name of the independent voltage or current source which is the noise input reference. **NUMS** is the summary interval. SPICE will compute the equivalent output noise at the specified output as well as the equivalent input noise at the specified input. In addition, the contributions of every noise generator in the circuit will be printed at every **NUMS** frequency points (the summary interval). If **NUMS** is zero, no summary printout will be made.

The output noise and the equivalent input noise may also be printed and/or plotted (see the description of the **.PRINT** and **.PLOT** cards).

### 9.13. **.TRAN Card**

General form:

**.TRAN TSTEP TSTOP <TSTART <TMAX>> <UIC>**

Examples:

**.TRAN 1NS 100NS**

**.TRAN 1NS 1000NS 500NS**

**.TRAN 10NS 1US UIC**

**TSTEP** is the printing or plotting increment for line-printer output. For use with the post-processor, **TSTEP** is the suggested computing increment. **TSTOP** is the final time, and **TSTART** is the initial time. If **TSTART** is omitted, it is assumed to be zero. The transient analysis always begins at time zero. In the interval  $<zero, TSTART>$ , the circuit is analyzed (to reach a steady state), but no outputs are stored. In the interval  $<TSTART, TSTOP>$ , the circuit is analyzed and outputs are stored. **TMAX** is the maximum stepsize that SPICE will use (for default, the program chooses either **TSTEP** or  $(TSTOP - TSTART)/50.0$ , whichever is smaller. **TMAX** is useful when one wishes to guarantee a computing interval which is smaller than the printer increment, **TSTEP**.

**UIC** (use initial conditions) is an optional keyword which indicates that the user does not want SPICE to solve for the quiescent operating point before beginning the transient analysis. If this keyword is specified, SPICE uses the values specified using **IC=...** on the various elements as the initial transient condition and proceeds with the analysis. If the **IC** card has been specified, then the node voltages on the **IC** card are used to compute the initial conditions for the devices. Look at the description on the **IC** card for its interpretation when **UIC** is not specified.

#### 9.14. **.FOUR** Card

General form:

**.FOUR FREQ OV1 < OV2 OV3 ...>**

Examples:

**.FOUR 100K V(5)**

This card controls whether SPICE performs a Fourier analysis as a part of the transient analysis. **FREQ** is the fundamental frequency, and **OV1, ...,** are the output variables for which the analysis is desired. The Fourier analysis is performed over the interval **< TSTOP-period, TSTOP>**, where **TSTOP** is the final time specified for the transient analysis, and period is one period of the fundamental frequency. The dc component and the first nine components are determined. For maximum accuracy, **TMAX** (see the **.TRAN** card) should be set to period/100.0 (or less for very high-Q circuits).

#### 9.15. **.PRINT** Cards

General form:

**.PRINT PRTYPE OV1 < OV2 ... OV8>**

Examples:

```
.PRINT TRAN V(4) I(VIN)
.PRINT AC VM(4,2) VR(7) VP(8,3)
.PRINT DC V(2) I(VSRC) V(23,17)
.PRINT NOISE INOISE
.PRINT DISTO HD3 SIM2(DB)
```

This card defines the contents of a tabular listing of one to eight output variables. **PRTYPE** is the type of the analysis (**DC**, **AC**, **TRAN**, **NOISE**, or **DISTO**) for which the specified outputs are desired. The form for voltage or current output variables is as follows:

**V(N1< ,N2> )** specifies the voltage difference between nodes **N1** and **N2**. If **N2** (and the preceding comma) is omitted, ground (0) is assumed. For the ac analysis, five additional outputs can be accessed by replacing the letter **V** by:

```
VR - real part
VI - imaginary part
VM - magnitude
VP - phase
VDB - 20*log10(magnitude)
```

AD-A146 444

VLSI DESIGN TOOLS REFERENCE MANUAL RELEASE 20(U)  
WASHINGTON UNIV SEATTLE DEPT OF COMPUTER SCIENCE  
AUG 84 TR-84-08-07 MDA903-82-C-0424

4/4

UNCLASSIFIED

F/G 9/5

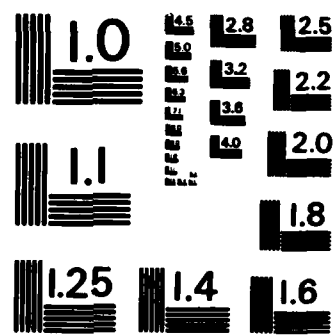
NL



END

FILMED

DTIC



MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A

**I(VXXXXXXX)** specifies the current flowing in the independent voltage source named **VXXXXXXX**. Positive current flows from the positive node, through the source, to the negative node. For the ac analysis, the corresponding replacements for the letter **I** may be made in the same way as described for voltage outputs.

Output variables for the noise and distortion analyses have a different general form from that of the other analyses, i.e.

**OV<(X)>**

where **OV** is any of **ONoise** (output noise), **INoise** (equivalent input noise), **D2**, **HD3**, **SIM2**, **DIM2**, or **DIM3** (see description of distortion analysis), and **X** may be any of:

**R** - real part  
**I** - imaginary part  
**M** - magnitude (default if nothing specified)  
**P** - phase  
**DB** -  $20\log_{10}(\text{magnitude})$

thus, **SIM2** (or **SIM2(M)**) describes the magnitude of the **SIM2** distortion measure, while **HD2(R)** describes the real part of the **HD2** distortion measure.

There is no limit on the number of **.PRINT** cards for each type of analysis.

#### 9.16. **.PLOT** Cards

General form:

**.PLOT PLTYPE OV1 <(PLO1,PHI1)> <OV2 <(PLO2,PHI2)> ... OV8>**

Examples:

```
.PLOT DC V(4) V(5) V(1)
.PLOT TRAN V(17,5) (2,5) I(VIN) V(17) (1,9)
.PLOT AC VM(5) VM(31,24) VDB(5) VP(5)
.PLOT DISTO HD2 HD3(R) SIM2
.PLOT TRAN V(5,3) V(4) (0,5) V(7) (0,10)
```

This card defines the contents of one plot of from one to eight output variables. **PLTYPE** is the type of analysis (**DC**, **AC**, **TRAN**, **NOISE**, or **DISTO**) for which the specified outputs are desired. The syntax for the **OVI** is identical to that for the **.PRINT** card, described above.

The optional plot limits (**PLO,PHI**) may be specified after any of the output variables. All output variables to the left of a pair of plot limits (**PLO,PHI**) will be plotted using the same lower and upper plot bounds. If plot limits are not specified, **SPICE** will automatically determine the minimum and maximum values of all output variables being plotted and scale the plot to fit. More than one scale will be used if the output variable values warrant (i.e., mixing output variables with values which are orders-of-magnitude different still gives readable plots).

The overlap of two or more traces on any plot is indicated by the letter **X**.

When more than one output variable appears on the same plot, the first variable specified will be printed as well as plotted. If a printout of all variables is desired, then a companion **.PRINT** card should be included.

There is no limit on the number of **.PLOT** cards specified for each type of analysis.

**10. APPENDIX A: EXAMPLE DATA DECKS****10.1. Circuit 1**

The following deck determines the dc operating point and small-signal transfer function of a simple differential pair. In addition, the ac small-signal response is computed over the frequency range 1Hz to 100MEGHz.

**SIMPLE DIFFERENTIAL PAIR**

```
VCC 7 0 12
VEE 8 0 -12
VIN 1 0 AC 1
RS1 1 2 1K
RS2 6 0 1K
Q1 3 2 4 MOD1
Q2 5 6 4 MOD1
RC1 7 3 10K
RC2 7 5 10K
RE 4 8 10K
.MODEL MOD1 NPN BF=50 VAF=50 IS=1E-12 RB=100 CJC=.5PF TF=.6NS
.TF V(5) VIN
.AC DEC 10 1 100MEG
.PLOT AC VM(5) VP(5)
.PRINT AC VM(5) VP(5)
.END
```

**10.2. Circuit 2**

The following deck computes the output characteristics of a MOS- FET device over the range 0-10V for VDS and 0-5V for VGS.

**MOS OUTPUT CHARACTERISTICS**

```
.OPTIONS NODE NOPAGE
VDS 3 0
VGS 2 0
M1 1 2 0 0 MOD1 L=4U W=6U AD=10P AS=10P
.MODEL MOD1 NMOS VTO=-2 NSUB=1.0E15 UO=550
* VIDS MEASURES ID, WE COULD HAVE USED VDS, BUT ID WOULD BE NEGATIVE
VIDS 3 1
.DC VDS 0 10 .5 VGS 0 5 1
.PRINT DC I(VIDS) V(2)
.PLOT DC I(VIDS)
.END
```

**10.3. Circuit 3**

The following deck determines the dc transfer curve and the transient pulse response of a simple RTL inverter. The input is a pulse from 0 to 5 Volts with delay, rise, and fall times of 2ns and a pulse width of 30ns. The transient interval is 0 to 100ns, with printing to be done every nanosecond.

```

SIMPLE RTL INVERTER
VCC 4 0 5
VIN 1 0 PULSE 0 5 2NS 2NS 2NS 30NS
RB 1 2 10K
Q1 3 2 0 Q1
RC 3 4 1K
.PLOT DC V(3)
.PLOT TRAN V(3) (0,5)
.PRINT TRAN V(3)
.MODEL Q1 NPN BF 20 RB 100 TF .1NS CJC 2PF
.DC VIN 0 5 0.1
.TRAN 1NS 100NS
.END

```

#### 10.4. Circuit 4

The following deck simulates a four-bit binary adder, using several subcircuits to describe various pieces of the overall circuit.

#### ADDER - 4 BIT ALL-NAND-GATE BINARY ADDER

##### \*\*\* SUBCIRCUIT DEFINITIONS

##### SUBCKT NAND 1 2 3 4

\* NODES: INPUT(2), OUTPUT, VCC

```

Q1 9 5 1 QMOD
D1CLAMP 0 1 DMOD
Q2 9 5 2 QMOD
D2CLAMP 0 2 DMOD
RB 4 5 4K
R1 4 6 1.6K
Q3 6 9 8 QMOD
R2 8 0 1K
RC 4 7 130
Q4 7 6 10 QMOD
DVBEDROP 10 3 DMOD
Q5 3 8 0 QMOD
.ENDS NAND

```

##### SUBCKT ONEBIT 1 2 3 4 5 6

\* NODES: INPUT(2), CARRY-IN, OUTPUT, CARRY-OUT, VCC

```

X1 1 2 7 6 NAND
X2 1 7 8 6 NAND
X3 2 7 9 6 NAND
X4 8 9 10 6 NAND
X5 3 10 11 6 NAND
X6 3 11 12 6 NAND
X7 10 11 13 6 NAND
X8 12 13 4 6 NAND
X9 11 7 5 6 NAND
.ENDS ONEBIT

```

```

SUBCKT TWOBIT 1 2 3 4 5 6 7 8 9
  * NODES: INPUT - BIT0(2) / BIT1(2), OUTPUT - BIT0 / BIT1,
  * CARRY-IN, CARRY-OUT, VCC
X1 1 2 7 5 10 9 ONEBIT
X2 3 4 10 6 8 9 ONEBIT
ENDS TWOBIT
SUBCKT FOURBIT 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
  * NODES: INPUT - BIT0(2) / BIT1(2) / BIT2(2) / BIT3(2),
  * OUTPUT - BIT0/BIT1/BIT2/BIT3,CARRY-IN,CARRY-OUT,VCC
X1 1 2 3 4 9 10 13 16 15 TWOBIT
X2 5 6 7 8 11 12 16 14 15 TWOBIT
ENDS FOURBIT

```

\*\*\* DEFINE NOMINAL CIRCUIT

```

MODEL DMOD D
MODEL QMOD NPN(BF=75 RB=100 CJE=1PF CJC=3PF)
VCC 99 0 DC 5V
VIN1A 1 0 PULSE(0 3 0 10NS 10NS 10NS 50NS)
VIN1B 2 0 PULSE(0 3 0 10NS 10NS 20NS 100NS)
VIN2A 3 0 PULSE(0 3 0 10NS 10NS 40NS 200NS)
VIN2B 4 0 PULSE(0 3 0 10NS 10NS 80NS 400NS)
VIN3A 5 0 PULSE(0 3 0 10NS 10NS 160NS 800NS)
VIN3B 6 0 PULSE(0 3 0 10NS 10NS 320NS 1600NS)
VIN4A 7 0 PULSE(0 3 0 10NS 10NS 640NS 3200NS)
VIN4B 8 0 PULSE(0 3 0 10NS 10NS 1280NS 6400NS)
X1 1 2 3 4 5 6 7 8 9 10 11 12 0 13 99 FOURBIT
RBIT0 9 0 1K
RBIT1 10 0 1K
RBIT2 11 0 1K
RBIT3 12 0 1K
RCOUT 13 0 1K
.PLOT TRAN V(1) V(2) V(3) V(4) V(5) V(6) V(7) V(8)
.PLOT TRAN V(9) V(10) V(11) V(12) V(13)
.PRINT TRAN V(1) V(2) V(3) V(4) V(5) V(6) V(7) V(8)
.PRINT TRAN V(9) V(10) V(11) V(12) V(13)

```

```

*** (FOR THOSE WITH MONEY (AND MEMORY) TO BURN)
.TRAN 1NS 6400NS

```

```

.OPTIONS ACCT LIST NODE LIMPTS=6401
END

```

### 10.5. Circuit 5

The following deck simulates a transmission-line inverter. Two transmission-line elements are required since two propagation modes are excited. In the case of a coaxial line, the first line (T1) models the inner conductor with respect to the shield, and the second line (T2) models the shield with respect to the out-side world.



**TRANSMISSION-LINE INVERTER**

```

V1 1 0 PULSE(0 1 0 0.1N)
R1 1 2 50
X1 2 0 0 4 TLINE
R2 4 0 50
SUBCKT TLINE 1 2 3 4
T1 1 2 3 4 Z0=50 TD=1.5NS
T2 2 0 4 0 Z0=100 TD=1NS
ENDS TLINE
.TRAN 0.1NS 20NS
PLOT TRAN V(2) V(4)
END

```

**11. APPENDIX B: NONLINEAR DEPENDENT SOURCES**

SPICE allows circuits to contain dependent sources characterized by any of the four equations

$$i=f(v) \quad v=f(v) \quad i=f(i) \quad v=f(i)$$

where the functions must be polynomials, and the arguments may be multidimensional. The polynomial functions are specified by a set of coefficients  $p_0, p_1, \dots, p_n$ . Both the number of dimensions and the number of coefficients are arbitrary. The meaning of the coefficients depends upon the dimension of the polynomial, as shown in the following examples:

Suppose that the function is one-dimensional (that is, a function of one argument). Then the function value  $f_v$  is determined by the following expression in  $a$  (the function argument):

$$f_v = p_0 + p_1 \cdot a + p_2 \cdot a^2 + p_3 \cdot a^3 + p_4 \cdot a^4 + p_5 \cdot a^5 + \dots$$

Suppose now that the function is two-dimensional, with arguments  $a$  and  $b$ . Then the function value  $f_v$  is determined by the following expression:

$$f_v = p_0 + p_1 \cdot a + p_2 \cdot b + p_3 \cdot a^2 + p_4 \cdot a \cdot b + p_5 \cdot b^2 + p_6 \cdot a^3 + p_7 \cdot a^2 \cdot b + p_8 \cdot a \cdot b^2 + p_9 \cdot b^3 + \dots$$

Consider now the case of a three-dimensional polynomial function with arguments  $a, b$ , and  $c$ . Then the function value  $f_v$  is determined by the following expression:

$$\begin{aligned}
f_v = & p_0 + p_1 \cdot a + p_2 \cdot b + p_3 \cdot c + p_4 \cdot a^2 + p_5 \cdot a \cdot b + \\
& p_6 \cdot a \cdot c + p_7 \cdot b^2 + p_8 \cdot b \cdot c + p_9 \cdot c^2 + p_{10} \cdot a^3 + \\
& p_{11} \cdot a^2 \cdot b + p_{12} \cdot a^2 \cdot c + p_{13} \cdot a \cdot b^2 + p_{14} \cdot a \cdot b \cdot c + \\
& p_{15} \cdot a \cdot c^2 + p_{16} \cdot b^3 + p_{17} \cdot b^2 \cdot c + \\
& p_{19} \cdot c^3 + p_{20} \cdot a^4 + \dots
\end{aligned}$$

Note: if the polynomial is one-dimensional and exactly one coefficient is specified, then SPICE assumes it to be  $p_1$  (and  $p_0 = 0.0$ ), in order to facilitate the input of linear controlled sources.

For all four of the dependent sources described below, the initial condition parameter is described as optional. If not specified, SPICE assumes 0 the initial condition for dependent sources is an initial initial condition to obtain the dc operating point of the circuit. After convergence has been obtained, the program continues iterating to obtain the exact value for the controlling variable. Hence, to reduce the computational effort for the dc operating point (or if the polynomial specifies a strong nonlinearity), a value fairly close to the actual controlling variable should be specified for the initial condition.

### 11.1. Voltage-Controlled Current Sources

General form:

**GXXXXXXX N+ N- <POLY(ND)> NC1+ NC1- ... P0 <P1 ...> <IC=...>**

Examples:

**G1 1 0 5 3 0 0.1M**  
**GR 17 3 17 3 0 1M 1.5M IC=2V**  
**GMLT 23 17 POLY(2) 3 5 1 2 0 1M 17M 3.5U IC=2.5, 1.3**

N+ and N- are the positive and negative nodes, respectively. Current flow is from the positive node, through the source, to the negative node. POLY(ND) only has to be specified if the source is multi-dimensional (one-dimensional is the default). If specified, ND is the number of dimensions, which must be positive. NC1+, NC1-, ... Are the positive and negative controlling nodes, respectively. One pair of nodes must be specified for each dimension. P0, P1, P2, ..., Pn are the polynomial coefficients. The (optional) initial condition is the initial guess at the value(s) of the controlling voltage(s). If not specified, 0.0 is assumed. The polynomial specifies the source current as a function of the controlling voltage(s). The second example above describes a current source with value

$$I = 10^{-3} \cdot V(27,3) + 1.5 \times 10^{-3} \cdot V(17,3)^2$$

note that since the source nodes are the same as the controlling nodes, this source actually models a nonlinear resistor.

### 11.2. Voltage-Controlled Voltage Sources

General form:

**EXXXXXXXX N+ N- <POLY(ND)> NC1+ NC1- ... P0 <P1 ...> <IC=...>**

Examples:

**E1 3 4 21 17 10.5 2.1 1.75**  
**EX 17 0 POLY(3) 13 0 15 0 17 0 0 1 1 1 IC=1.5,2.0,17.35**

N+ and N- are the positive and negative nodes, respectively. POLY(ND) only has to be specified if the source is multi-dimensional (one-dimensional is the default). If specified, ND is the number of dimensions, which must be positive. NC1+, NC1-, ... are the positive and negative controlling nodes, respectively. One pair of nodes must be specified for each dimension. P0, P1, P2, ..., Pn are the polynomial coefficients. The (optional) initial condition is the initial guess at the value(s) of the controlling voltage(s). If not specified, 0.0 is assumed. The polynomial specifies the source voltage as a function of the controlling voltage(s). The second example above describes a voltage source with value

$$V = V(13,0) + V(15,0) + V(17,0)$$

(in other words, an ideal voltage summer).

### 11.3. Current-Controlled Current Sources

General form:

**FXXXXXXX N+ N- <POLY(ND)> VN1 <VN2 ...> P0 <P1 ...> <IC=...>**

**Examples:**

```
F1 12 10 VCC 1MA 1.3M
FXFER 13 20 VSENS 0 1
```

N+ and N- are the positive and negative nodes, respectively. Current flow is from the positive node, through the source, to the negative node. POLY(ND) only has to be specified if the source is multi-dimensional (one-dimensional is the default). If specified, ND is the number of dimensions, which must be positive. VN1, VN2, ... are the names of voltage sources through which the controlling current flows; one name must be specified for each dimension. The direction of positive controlling current flow is from the positive node, through the source, to the negative node of each voltage source. P0, P1, P2, ..., Pn are the polynomial coefficients. The (optional) initial condition is the initial guess at the value(s) of the controlling current(s) (in Amps). If not specified, 0.0 is assumed. The polynomial specifies the source current as a function of the controlling current(s). The first example above describes a current source with value

$$I = 10^{-3} + 1.3 \times 10^{-3} \cdot (VCC)$$

#### 11.4. Current-Controlled Voltage Sources

**General form:**

```
HXXXXXXX N+ N- <POLY(ND)> VN1 <VN2 ...> P0 <P1 ...> <IC=...>
```

**Examples:**

```
HXY 13 20 POLY(2) VIN1 VIN2 0 0 0 1 IC=0.5 1.3
HR 4 17 VX 0 0 1
```

N+ and N- are the positive and negative nodes, respectively. POLY(ND) only has to be specified if the source is multi-dimensional (one-dimensional is the default). If specified, ND is the number of dimensions, which must be positive. VN1, VN2, ... are the names of voltage sources through which the controlling current flows; one name must be specified for each dimension. The direction of positive controlling current flow is from the positive node, through the source, to the negative node of each voltage source. P0, P1, P2, ..., Pn are the polynomial coefficients. The (optional) initial condition is the initial guess at the value(s) of the controlling current(s) (in Amps). If not specified, 0.0 is assumed. The polynomial specifies the source voltage as a function of the controlling current(s). The first example above describes a voltage source with value

$$V = I(VIN1) - I(VIN2)$$

#### 12. APPENDIX C: BIPOLAR MODEL EQUATIONS

Acknowledgment: This section has been contributed by Bill Bidermann at HP labs.

(G<sub>min</sub> terms omitted)

##### 12.1 D.C. MODEL

$$I_C = \frac{I_S}{Q_B} \left( e^{\frac{q(V_{BE})}{kT}} - e^{\frac{q(V_{BC})}{kT}} \right) - \frac{I_S}{B_F} \left( e^{\frac{q(V_{BC})}{kT}} - 1 \right) - I_{SC} \left( e^{\frac{q(V_{BC})}{kT}} - 1 \right)$$

$$I_B = \frac{I_S}{B_F} \left( e^{\frac{q(V_{BE})}{kT}} - 1 \right) + \frac{I_S}{B_R} \left( e^{\frac{q(V_{BC})}{kT}} - 1 \right) + I_{SE} \left( e^{\frac{q(V_{BE})}{kT}} - 1 \right) + I_{SC} \left( e^{\frac{q(V_{BC})}{kT}} - 1 \right)$$

NOTE: The last two terms in the expression of the base current  $I_B$  represent the components due to recombination in the BE and BC space charge regions at low injection.

If  $I_{RB}$  not specified

$$R_{BB} = R_{BM} + \frac{R_B - R_{BM}}{Q_B}$$

If  $I_{RB}$  specified

$$R_{BB} = 3(R_B - R_{BM}) \left[ \frac{\tan(z)}{z \tan^2(z)} - z + R_{BM} \right]$$

Where:

$$z = \frac{-1 + \sqrt{1 + 144 I_B / (I_{RB}^2 + 1)}}{24 / \pi^2 \sqrt{I_B / I_{RB}}}$$

$$Q_B = \frac{Q_1}{2} (1 + \sqrt{1 + 4 Q_2})$$

$$Q_1 = \frac{-1}{1 - \frac{V_{BC}}{V_{AF}} - \frac{V_{BE}}{V_{AR}}}$$

$$Q_2 = \frac{I_S}{I_{KF}} (e^{(N_F) \frac{q V_{BE}}{k T}} - 1) + \frac{I_S}{I_{KR}} (e^{(N_R) \frac{q V_{BC}}{k T}} - 1)$$

NOTE:  $I_{RB}$  is the current where the base resistance falls halfway to its minimum value.  $V_{AF}$  and  $V_{AR}$  are forward and reverse Early voltages respectively.  $I_{KF}$  and  $I_{KR}$  determine the high current beta rolloff with  $I_C$ .  $I_{SE}$ ,  $I_{SC}$ ,  $N_E$  and  $N_C$  determine the low current beta rolloff with  $I_C$ .

## 12.2 A.C. MODEL

$$C_{BE} = \frac{C_{JE}}{V_{BE}} \left[ (TFF \frac{I_S}{Q_B} (e^{(N_F) \frac{q V_{BE}}{k T}} - 1) + C_{JE} (1 - \frac{V_{BE}}{V_{JE}})^{-M_J} \right]$$

Where:

$$TFF = TF (1 + XTF \cdot (\frac{I_F}{I_F + TFF^2}) \cdot e^{1.44 \frac{V_{BC}}{V_{TF}}})$$

$$I_F = I_S (e^{(N_F) \frac{q V_{BE}}{k T}} - 1)$$

$$C_{BI} = C_{BC} (1 - X_{CJC})$$

$$C_{B2} = C_{BC} \cdot X_{CJC}$$

$$C_{BC} = TR \left( \frac{I_S}{(N_R) k T} e^{(N_R) \frac{q V_{BC}}{k T}} \right) + C_{JC} (1 - \frac{V_{BC}}{V_{JC}})^{-M_{JC}}$$

$$C_{SS} = C_{JS} (1 - \frac{V_{CS}}{V_{JS}})^{-M_{JS}}$$

NOTE: all junction capacitances of the form  $C_0 \cdot (1 - \frac{V}{V_0})^{-M}$  revert to the form

$$(1 - \frac{C_0}{FC})^M (1 + \frac{M(V - FC)}{(1 - \frac{C_0}{FC})})$$

when  $V > FC$  (For  $C_{SS}$  assumes  $FC = 0$ )

## 12.3 NOISE MODEL

Thermal Noise :

$$I_{RBB}^2 = \frac{4kT}{R} B \Delta f$$

$$I_{RC}^2 = \frac{4kT}{R_C} \Delta f$$

$$I_{RE}^2 = \frac{4kT}{2R_{ED}} \Delta f + K_F \frac{I_B \cdot A_F}{I} \Delta f$$

Note: The first two terms are shot noise and the last term is flicker noise.

$$ICN^2 = 2qICA f$$

Note: This is shot noise.

## 12.4 TEMPERATURE EFFECTS

All junctions have dependences identical to the diode model but all N factors are considered equal 1.

BF and BR go as  $(\frac{T}{TNOM})^{XTB}$  when NF=1. This is done through appropriate changes in BF, BR and ISE, ISC according to the following equations respectively:

$$\overline{BF} = BF \cdot (\frac{T}{TNOM})^{XTB}$$

$$\overline{BR} = BR \cdot (\frac{T}{TNOM})^{XTB}$$

$$\overline{ISE} = ISE \cdot (\frac{T}{TNOM})^{(XTI - XTB)} \cdot e^{\frac{qEG}{nk} \cdot \frac{T - TNOM}{T \cdot TNOM}}$$

$$\overline{ISC} = ISC \cdot (\frac{T}{TNOM})^{(XTI - XTB)} \cdot e^{\frac{qEG}{nk} \cdot \frac{T - TNOM}{T \cdot TNOM}}$$

## 12.5 EXCESS PHASE

This is a delay (linear phase) in the gm generator in AC analysis. It is also used in transient analysis using a Bessel polynomial approximation. Excess phase, PTF, is specified as the number of extra degrees of phase at the frequency

$$f = \frac{1}{2\pi} \cdot \frac{1}{PTFHertz}$$

## 12. APPENDIX D: ALTER STATEMENT AND THE SOURCE-STEPPING METHOD

The ALTER statement allows SPICE to run with altered circuit parameters.

General form:

```
.ALTER
ELEMENT CARDS (DEVICE CARDS, MODEL CARDS)
.ALTER (or .END CARD)
```

Examples:

```
R1 1 0 5K
VCC 3 0 10
M1 3 2 0 MOD1 L=5U W=2U
MODEL MOD1 NMOS(VTO=1.0 KP=2.0E-5 PHI=0.6 NSUB=2.0E15 TOX=0.1U)
.ALTER
R1 1 0 3.5K
VCC 3 0 12
M1 3 2 0 MOD1 L=10U W=2U
MODEL MOD1 NMOS(VTO=1.2 KP=2.0E-5 PHI=0.6 NSUB=5.0E15 TOX=1.5U)
.ALTER
M1 3 2 0 MOD1 L=10U W=4U
.END
```

This card introduces the element(s), device(s) and model(s) whose parameters are changed during the execution of the input deck. The analyses specified in the deck will start

over again with the changed parameters. The `.ALTER` card with the cards defining the new parameters should be placed just before the `.END` card. The syntax for the element (device, model) cards is identical to that of the cards with the original parameters.

There is no limit on the number of `.ALTER` cards and the circuit will be re-analyzed as many times as the number of `.ALTER` cards. Subsequent `.ALTER` operations employ parameters of the previous change. No topological change of the circuit is allowed.

The source-stepping method can enhance DC convergence. But it is slower than direct use of the Newton-Raphson method. Therefore it is best used as an alternative to achieve convergence of DC operating point when the circuit fails to converge by using the Newton-Raphson method. The source-stepping method is used by SPICE when the variable `ITL6` in the `.OPTIONS` card is set to the iteration limit at each step of the source(s).